国内首本介绍底层硬件驱动和上层应用程序贯通设计的著作，拥有丰富的实验程序、完整的项目实例，帮助读者快速提高嵌入式硬件与Windows CE的开发能力。

单片机与嵌入式

*Windows CE Embedded System Development (Based on ARM11)*

# Windows CE 嵌入式系统程序开发（基于ARM11）

胡文 冯剑 姜海涛 胡玥 编著

机械工业出版社
China Machine Press

在当今软件开发人员中，在Windows上用微软开发工具的人员占有相当大的优势，开发人员只要熟悉任一Windows平台，只要用上一点点时间，就会马上熟悉Windows CE平台的开发，90%以上的API函数调用都完全一样。基于这样一个巨大的优势，任何一个企业可以支付较少的人力成本找到合适的工程师，快速完成产品开发。

本书着重讲解了Windows CE体系结构和搭建Windows Embedded CE 6.0应用程序的开发环境，以及使用Windows CE窗体控件开发应用程序。在Windows Embedded CE 6.0和Visual Studio 开发环境下，采用C++开发接口驱动程序。本书提供完整的综合应用项目的程序开发实例，介绍了功能模块设计和数据库设计，详细讲解了开发过程和原始程序代码。读者可以深刻理解和掌握嵌入式系统开发的整个过程，了解底层驱动程序驱动硬件的原理和上层应用程序的设计方法，真正做到底层驱动开发与上层应用程序开发相结合。

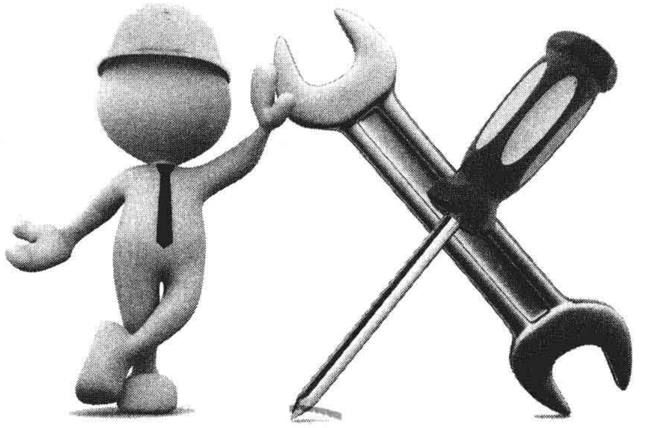
上架指导：单片机及嵌入式

ISBN 978-7-111-41915-0

定价：69.00元

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com



Windows CE Embedded System Development (Based on ARM11)

# Windows CE 嵌入式系统程序开发

## (基于ARM11)

安徽大学图书馆
藏书
www.lib.ahu.edu.cn

胡文 冯剑 姜海涛 胡玥 编著

机械工业出版社
China Machine Press

**图书在版编目（CIP）数据**

Windows CE嵌入式系统程序开发（基于ARM11）/胡文等编著. —北京：机械工业出版社，2013.4
（单片机与嵌入式丛书）

ISBN 978-7-111-41915-0

Ⅰ.W… Ⅱ.胡… Ⅲ.Windows操作系统-程序设计 Ⅳ.TP316.7

中国版本图书馆CIP数据核字（2013）第057845号

**版权所有·侵权必究**
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

本书主要分为3部分，包括软件篇、硬件篇和项目篇。软件篇着重讲解了Windows CE体系结构、搭建Windows Embedded CE 6.0开发环境和Windows CE应用程序开发；硬件篇重点讲解了S3C6410硬件体系结构、Windows CE Boot Loader与OAL开发，在Windows Embedded CE 6.0和Visual Studio 2005开发环境下，采用C++开发接口驱动程序；项目篇提供完整的综合应用项目的程序开发实例，详细介绍了开发过程和源代码。并且安排了丰富的实验内容与课后实践，让读者深刻理解和掌握嵌入式系统开发的整个过程，了解底层驱动程序驱动硬件的原理和上层应用程序的设计方法。

本书内容丰富实用，层次清晰，叙述详尽，方便教学与自学。本书可作为高等院校计算机类、电子类和控制类等专业高年级本科生、研究生学习嵌入式Windows CE程序开发的教材，也可作为全国大学生电子设计竞赛培训教材，以及工程技术人员进行嵌入式系统开发与应用的参考书。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：王 彬 秦 健
北京诚信伟业印刷有限公司印刷
2013年5月第1版第1次印刷
186mm×240mm·30印张
标准书号：ISBN 978-7-111-41915-0
定 价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

# 前　言

本书以嵌入式操作系统Windows Embedded CE 6.0和ARM11微处理器S3C6410为基础，在软件篇着重讲解了Windows CE体系结构和搭建Windows Embedded CE 6.0应用程序的开发环境，以及使用Windows CE窗体控件开发应用程序。硬件篇介绍了S3C6410的硬件结构，针对相应的硬件目标平台开发Boot Loader和OAL，在Windows Embedded CE 6.0和Visual Studio 2005开发环境下，采用C++开发接口驱动程序。项目篇提供完整的综合应用项目的程序开发实例，介绍了功能模块设计和数据库设计，详细讲解了开发过程和源代码。本书还安排了丰富的实验内容与课后实践，让读者深刻理解和掌握嵌入式系统开发的整个过程，了解底层驱动程序驱动硬件的原理和上层应用程序的设计方法，真正做到了底层驱动的开发与上层应用程序的开发相结合。

本书内容丰富实用、叙述详尽，方便教学与自学。通过介绍结合了DMA-6410XP平台的实验程序，有利于读者掌握嵌入式系统程序开发的方法，并能培养读者综合分析、开发创新和工程设计的能力。通过对本书的学习，读者可以快速提高Windows CE的编程能力和实际开发水平。

全书分三部分，共9章。

**第一部分：软件篇**

第1章介绍了Windows Embedded CE 6.0体系结构，以及Windows CE系统的内存管理、存储管理与文件系统、进程、线程和中断处理等内容。

第2章介绍了在PC上如何搭建应用程序的开发环境，包括Visual Studio 2005开发工具的安装、Windows Embedded CE 6.0的安装、定制DMA-6410XP系统、执行NK.nb0和Windows CE烧写等。

第3章介绍了本书中第一个Windows CE程序和界面程序设计，使用Windows CE窗体控件创建UI、安装和连接配置SQLCE，以及OLEDB编程访问SQLCE数据库和SQLCE远程数据访问方法。

**第二部分：硬件篇**

第4章介绍了S3C6410的硬件结构，包括S3C6410微处理器、GPIO输入输出接口、PWM定时器、DMA 控制器、URAT串行接口、IIC总线接口、SPI接口，并举例编写接口电路驱动程序。

第5章介绍了Boot Loader的基本概念、典型结构与启动流程、Boot Loader的实现过程，以及OAL的功能与构成、启动流程。针对相应的硬件目标平台开发Boot Loader，然后介绍了OAL

的开发方法。

第6章在Windows Embedded CE 6.0和Visual Studio 2005开发环境下，采用C++编写接口驱动程序。主要内容包括LED接口及驱动程序、键盘接口及驱动程序、ADC接口及驱动程序、UART接口及驱动程序、GPRS接口及驱动程序、直流电动机接口及驱动程序、步进电动机接口及驱动程序、蓝牙接口及驱动程序等。

**第三部分：项目篇**

第7章介绍了GPS相关知识、搭建开发环境的方法、采用C++开发GPS定位程序的方法和开发GPS与Google Earth定位程序的方法。

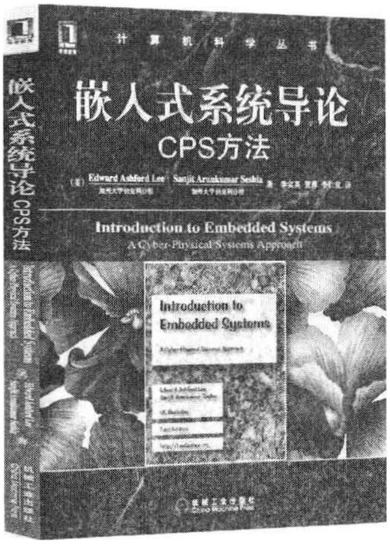
第8章从总体设计着手，介绍了功能模块设计和数据库设计的方法，以及采用C++开发数据传输模块程序和鲜奶配送终端应用程序的方法。

第9章介绍了搭建开发环境和初始化调用环境的方法，以及采用C++开发无线点餐终端应用程序的方法。

本书由胡文教授拟订大纲和目录，并编写了第4章和第6章；冯剑编写了第2章和第3章；姜海涛编写了第1章、第5章和第8章；胡玥编写了第7章、第9章。哈尔滨商业大学的陈铭、李杨、赵艳丽、张凯、陈楠等人为本书的编写做了大量的工作，在此一并表示衷心感谢。本书在编写过程中，参考了大量国内外著作和资料，得到了许多专家和学者的大力支持，听取了多方面的宝贵意见和建议，在此对他们表示衷心感谢。

由于时间仓促和水平所限，本书难免有疏漏和不足之处，敬请各位读者批评指正，以期再版时修订。作者邮箱：huwen1957@126.com。

# 推荐阅读

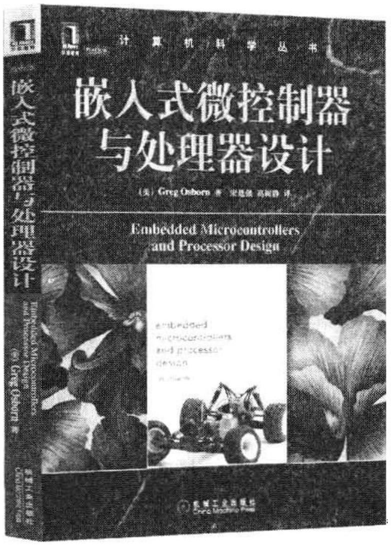



**嵌入式系统导论：CPS方法**

作者：Edward Ashford Lee 等 ISBN：978-7-111-36021-6 定价：55.00元

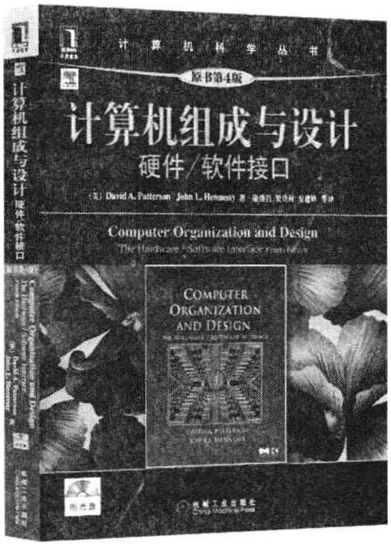
**嵌入式计算系统设计原理（第2版）**

作者：Wayne Wolf ISBN：978-7-111-27068-3 定价：55.00元




**嵌入式微控制器与处理器设计**

作者：Greg Osborn ISBN：978-7-111-32281-8 定价：59.00元




**计算机组成与设计：硬件/软件接口（原书第4版）**

作者：David A. Patterson 等 ISBN：978-7-111-35305-8 定价：99.00元

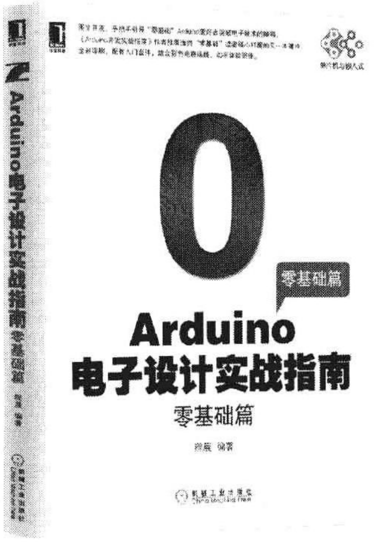
# 推 荐 阅 读

**嵌入式系统软硬件协同设计实战指南：基于Xilinx Zynq**

作者：陆佳华 等 ISBN：978-7-111-41107-9 定价：69.00元

**兼容ARM9的软核处理器设计：基于FPGA**

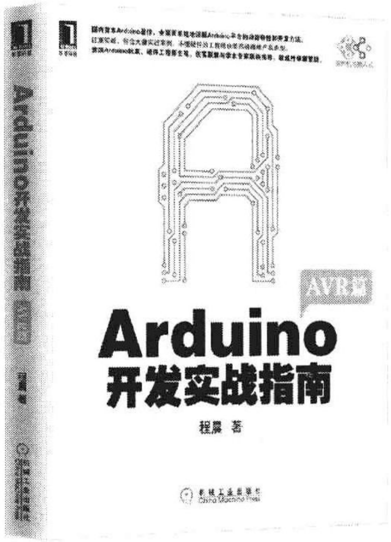
作者：李新兵 ISBN：978-7-111-37572-2 定价：69.00元




**Ar… ：零基础篇**

作… 定价：59.00元

北航 C1640737




**Arduino开发实战指南：AVR篇**

作者：程晨 ISBN：978-7-111-37005-5 定价：59.00元

# 目　录

前言

## 第一部分　软件篇

### 第1章　Windows CE系统结构 / 2

1.1　Windows Embedded CE 6.0的体系结构 / 2
1.2　内存管理 / 7
1.2.1　内存结构 / 7
1.2.2　Windows Embedded CE 6.0虚拟内存模型 / 8
1.3　文件系统与存储管理 / 12
1.3.1　文件系统 / 12
1.3.2　注册表 / 14
1.3.3　存储管理器 / 16
1.4　进程与线程 / 19
1.4.1　Windows CE进程 / 19
1.4.2　Windows CE线程 / 21
1.4.3　线程调度 / 24
1.4.4　线程同步 / 26
1.4.5　进程间通信 / 30
1.5　中断处理 / 31
1.5.1　中断基本概念 / 31
1.5.2　中断服务例程 / 33
1.5.3　中断服务线程 / 34

### 第2章　搭建Windows CE开发环境 / 36

2.1　安装Visual Studio 2005 / 36
2.1.1　安装Visual Studio 2005 / 36
2.1.2　安装Visual Studio 2005 Service Pack 1 / 40
2.2　安装Windows Embedded CE 6.0 / 42
2.2.1　安装Platform Builder 6.0 Service Pack 1 / 47
2.2.2　安装Windows Embedded CE 6.0 R2 / 50
2.2.3　安装Windows Embedded CE 6.0 R3 / 53
2.3　定制DMA-6410 XP系统 / 55
2.3.1　使用项目向导 / 55
2.3.2　构建新的平台 / 57
2.3.3　编辑和修改Windows Embedded CE项目 / 62
2.3.4　添加必要的功能 / 63
2.3.5　Build / 67
2.4　执行NK.nb0 / 70
2.5　Windows CE烧写 / 73

### 第3章　Windows CE应用程序开发 / 85

3.1　第一个WinCE程序 / 85
3.1.1　SDK的创建与安装 / 85
3.1.2　应用程序设计 / 87
3.1.3　使用C#编写LED驱动程序 / 91
3.1.4　使用MFC编写界面程序 / 93
3.2　窗体基本控件（MFC）/ 102
3.2.1　控件的属性、使用方法以及

事件 / 103
3.2.2 Edit控件 / 106
3.2.3 CheckBox控件 / 108
3.2.4 RadioButton控件 / 109
3.2.5 ComboBox控件 / 113
3.2.6 ListBox控件 / 119
3.2.7 Spin控件 / 124
3.2.8 Slider控件 / 128
3.3 SQL Server CE / 131
3.3.1 SQLCE简介 / 131
3.3.2 SQLCE 3.0的安装 / 132
3.3.3 SQLCE 3.0连接配置 / 134
3.4 OLEDB编程访问SQLCE 3.0 / 140
3.4.1 OLEDB概述 / 140
3.4.2 OLEDB编程 / 141
3.5 SQLCE远程数据访问 / 151
3.5.1 拉数据 / 152
3.5.2 推数据 / 153
3.5.3 远程T-SQL操作 / 154
3.6 远程数据访问综合示例 / 154

# 第二部分 硬件篇

## 第4章 S3C6410硬件体系结构 / 170

4.1 S3C6410微处理器 / 170
4.1.1 S3C6410结构图 / 170
4.1.2 S3C6410处理器特性 / 170
4.2 GPIO输入输出接口 / 173
4.2.1 GPIO输入输出接口简介 / 173
4.2.2 GPIO专用寄存器 / 174
4.2.3 GPIO接口应用举例 / 183
4.3 PWM定时器 / 185
4.3.1 PWM定时器简介 / 185
4.3.2 PWM的操作 / 185
4.3.3 PWM专用寄存器 / 187
4.3.4 定时器应用举例 / 192
4.4 DMA控制器 / 195
4.4.1 DMA控制器简介 / 195
4.4.2 DMA专用寄存器 / 198
4.4.3 DMA控制器应用举例 / 203
4.5 UART串行接口 / 207
4.5.1 UART串行接口简介 / 207
4.5.2 UART的操作 / 208
4.5.3 UART专用寄存器 / 209
4.5.4 UART接口应用举例 / 215
4.6 I2C总线接口 / 219
4.6.1 I2C总线接口简介 / 219
4.6.2 I2C总线接口操作模式 / 220
4.6.3 I2C总线接口专用寄存器 / 222
4.6.4 I2C总线接口应用举例 / 225
4.7 SPI接口 / 228
4.7.1 SPI总线接口简介 / 228
4.7.2 SPI操作 / 228
4.7.3 SPI专用寄存器 / 229
4.7.4 SPI接口应用举例 / 234
4.8 DMA-6410XP实验平台 / 235
4.8.1 实验平台功能 / 235
4.8.2 外观及接口 / 236
4.8.3 平台硬件配置 / 236
4.8.4 支持操作系统 / 239

## 第5章 Windows CE Boot Loader与OAL / 241

5.1 Boot Loader介绍 / 241
5.1.1 Boot Loader的引导选项 / 241
5.1.2 Boot Loader的典型特征 / 242

5.1.3 Windows CE中Boot Loader为可选项 / 243
5.2 Windows Boot Loader的软件框架和支持库 / 244
5.3 Windows CE操作系统映像文件格式 / 248
5.4 Windows CE的系统启动执行 / 253
5.4.1 OAL的功能 / 253
5.4.2 OAL的构成 / 254
5.4.3 OAL的启动 / 255
5.5 DMA-6410XP中Windows CE OAL开发 / 265
5.5.1 OEMGLOBAL结构体及其整体变量 / 266
5.5.2 OEM地址映像表 / 267
5.5.3 OEMIoControl函数 / 268
5.5.4 OAL的Cache操作OEM函数 / 269

**第6章 接口驱动程序开发 / 271**

6.1 驱动架构分析 / 271
6.1.1 驱动程序分类 / 271
6.1.2 流接口驱动程序架构 / 272
6.1.3 添加一个流驱动到BSP中 / 275
6.2 LED接口及驱动程序 / 276
6.2.1 LED电路设计 / 277
6.2.2 S3C6410GPIO专用寄存器 / 277
6.2.3 LED驱动程序分析 / 278
6.2.4 LED底层驱动程序设计 / 279
6.2.5 LED驱动程序设计 / 282
6.3 键盘接口及驱动程序 / 293
6.3.1 键盘概述 / 293
6.3.2 矩阵式键盘 / 294
6.3.3 接口专用寄存器 / 296
6.3.4 键盘驱动操作分析 / 298
6.3.5 键盘底层驱动程序设计 / 301
6.3.6 键盘驱动程序设计 / 304
6.4 ADC接口及驱动程序 / 308
6.4.1 S3C6410 ADC / 308
6.4.2 A/D转换接口电路 / 309
6.4.3 ADC专用寄存器 / 311
6.4.4 A/D转换驱动分析 / 315
6.4.5 A/D转换底层驱动程序设计 / 316
6.4.6 A/D转换程序设计 / 320
6.5 UART串口驱动程序 / 323
6.5.1 串行接口概述 / 324
6.5.2 UART接口电路 / 325
6.5.3 UART专用寄存器 / 327
6.5.4 串行通信驱动程序分析 / 331
6.5.5 串行通信底层驱动程序设计 / 332
6.5.6 串口通信程序设计 / 342
6.6 GPRS接口及驱动程序 / 349
6.6.1 GPRS/GSM工作原理 / 349
6.6.2 AT命令 / 350
6.6.3 电话程序设计 / 350
6.6.4 短信程序设计 / 356
6.7 直流电动机接口及驱动程序 / 364
6.7.1 直流电动机驱动电路 / 364
6.7.2 PWM定时器专用寄存器 / 366
6.7.3 直流电动机底层驱动程序设计 / 368
6.7.4 直流电动机驱动程序设计 / 373
6.8 步进电动机接口及驱动程序 / 374

6.8.1 步进电动机驱动电路 / 374
6.8.2 控制寄存器 / 376
6.8.3 步进电动机底层驱动程序设计 / 377
6.8.4 步进电动机驱动程序设计 / 383
6.9 蓝牙接口及驱动程序 / 384
6.9.1 蓝牙相关知识 / 384
6.9.2 蓝牙模块的AT命令 / 386
6.9.3 消息 / 388
6.9.4 蓝牙驱动程序设计 / 388

# 第三部分 项目篇

## 第7章 GPS与Google Earth定位系统 / 398

7.1 GPS相关知识 / 398
7.1.1 卫星星座 / 398
7.1.2 地面监控系统 / 399
7.1.3 信号接收机 / 399
7.2 搭建环境 / 400
7.2.1 解析GPS定位信息 / 400
7.2.2 下载并安装Google Earth / 400
7.3 GPS定位程序 / 401
7.4 GPS与Google Earth定位程序 / 407
7.4.1 Google Earth_Start程序的建立 / 407
7.4.2 地图定位功能 / 411
7.4.3 在地图上新增地标 / 414

## 第8章 鲜奶配送系统 / 417

8.1 系统总体设计 / 417
8.2 搭建开发环境 / 418
8.3 系统数据库设计 / 424
8.4 系统初始化 / 425
8.5 终端应用程序设计 / 426
8.5.1 登录界面 / 426
8.5.2 主界面 / 428
8.5.3 配送模块 / 429
8.5.4 数据传输模块 / 441
8.5.5 历史数据查询 / 444
8.5.6 配送员设置 / 446

## 第9章 无线点餐系统 / 447

9.1 搭建开发环境 / 447
9.1.1 服务器端SQL Server 2005数据库的安装 / 448
9.1.2 SQLCE数据库的安装与配置 / 448
9.1.3 SQLCE 3.0连接配置 / 449
9.1.4 OLEDB编程访问SQLCE3.0 / 453
9.2 初始化调用环境 / 460
9.3 终端应用程序设计 / 461
9.3.1 系统的登录功能 / 461
9.3.2 系统主菜单 / 462
9.3.3 点餐功能 / 462
9.3.4 结算功能 / 464
9.3.5 查台功能 / 464
9.3.6 转台功能 / 464
9.3.7 更新功能 / 465
9.4 系统数据库设计 / 470

## 参考文献 / 472

# 第一部分　软件篇


◆ 第1章　Windows CE系统结构

◆ 第2章　搭建Windows CE开发环境

◆ 第3章　Windows CE应用程序开发

# 第1章　Windows CE系统结构

## 1.1　Windows Embedded CE 6.0的体系结构

Windows Embedded CE 6.0的体系结构如图1-1所示。

图1-1　Windows Embedded CE 6.0的体系结构

Windows CE 5.0的体系结构是一种分层式结构，从底层向上分别为硬件层、OEM层、操作系统层和应用层。而Windows Embedded CE 6.0划分为用户模式（User Mode）层和内核模式（Kernel Mode）层，其中CoreDLL等DLL同时出现在两个层中，驱动程序也可以加入到内核模式层中，以前的.exe可执行模块基本上都变成了.dll模块。Windows Embedded CE 6.0与之前的Windows CE 5.0体系结构的变化相对较大。

Windows CE 5.0被设计成一种围绕服务而存在的用户模式的程序，称为PSL（Process Server Library，程序服务库），其中NK.exe在内核模式下执行，而操作系统的其他部分则像单独的程序各自执行在用户模式下，例如文件系统Filesys.exe、图形窗口和事件子系统GWES.exe、驱动管理器Device.exe。这样分开的设计让操作系统更加健壮，但为整个操作系统提供主

要功能的服务提供者却以不同程序的身份出现。如果要使用某操作系统提供的服务，则至少会发生一次程序调用，甚至一个简单的操作系统提供的函数调用都不例外。这种设计对系统的效率影响是比较大的。

而Windows Embedded CE 6.0却不同，它将所有操作系统需要提供的服务转移到系统内核的虚拟机（Kernel’s Virtual Machine）。这样做的好处是，当系统程序调用服务时，服务程序已经变成了内核的一部分。然而这么做也造成了一些不稳定的机制，例如将驱动程序加载到内核等。Windows Embedded CE 6.0默认在内核模式下执行驱动程序，虽然这样做提高了系统的效率，但如果驱动程序不稳定，将对系统的整体稳定性产生非常严重的影响。当然，并不是所有的驱动程序都是在内核模式下执行的，在Windows Embedded CE 6.0安装完成之后，有一部分驱动程序就是在用户模式下执行的，这样更有利于系统的安全，但是以牺牲设备的性能为代价的。图1-2展示了Windows Embedded CE 6.0里的系统模块。

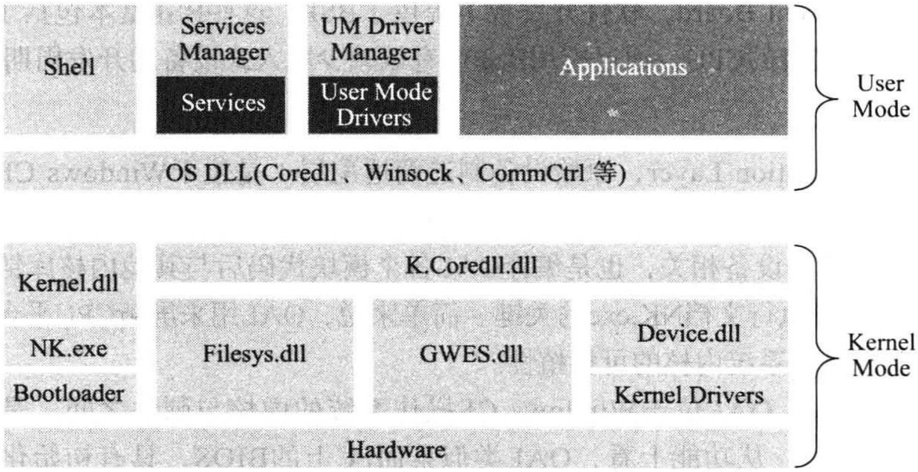



图1-2 Windows Embedded CE 6.0系统模块

如图1-2所示，Windows CE 5.0的各种系统模块，例如Filesys.exe、Device.exe、GWES.exe等，在Windows Embedded CE 6.0中变成了Filesys.dll、Device.dll、GWES.dll，只有NK.exe还是原来的名字。在Windows Embedded CE 6.0中这些服务已经成为系统程序调用（system call）。虽然NK.exe的名字没有变，但其已经区别于Windows CE 5.0中的NK.exe了。Windows CE 5.0中NK.exe提供的各种功能将由Kernel.dll来替代，NK.exe中仅包含一些OAL程序和保证兼容性的程序，这样做的好处是可使OEM和ISV厂商开发的程序与微软提供的Windows Embedded CE 6.0的程序分离，使得内核程序的升级更加容易、更加方便。

从以上叙述中可以看出，Windows CE系统的架构变化是很大的，这对Windows CE的二次开发厂商来说并不是什么好消息，因为这意味着很多建立在旧版本上的驱动，必须进行很大规模的重新设计。但微软公司强调这个过程是非常简单的，因为微软公司设计了一个补救措施。这个补救措施是通过图1-2中所示的处在内核模式的K.Coredll.dll来实现的，K.Coredll.dll名字与Coredll.dll非常相似，事实上它就是为了模拟Coredll.dll。K.Coredll.dll仍然在用户模式下执行，

当内核模式下的程序代码调用某些在Windows CE 5.0中处于用户模式下而在Windows Embedded CE 6.0中已转移到内核模式下的API时，K.Coredll.dll会调用这个程序，请求转向内核模式中相应的DLL。K.Coredll.dll并不是唯一有这种机制的DLL，只要其他的DLL需要，也可以采用这种策略。接下来将简要介绍Windows Embedded CE 6.0的系统模块。

### 1. 硬件

硬件层是指由CPU、存储器、I/O端口、扩展板卡等组成的嵌入式硬件系统，是Windows CE操作系统必不可少的载体。在嵌入式系统领域，因为设备制造商都是由不同领域、不同应用的厂商发展而来的，所以硬件结构相对复杂，仅CPU体系结构就有多种，Windows Embedded CE 6.0支持四种CPU体系结构（ARM、MIPS、x86、SHx）。

硬件的复杂性会增加程序开发的难度，因此出现了BSP（Board Support Package，板级支持包），以此来解决硬件体系结构的差别问题。Platform Builder for CE 6.0为很多常用的SDB（Software Development Board，软件开发板）提供了BSP，这些BSP基本包括了所有Windows Embedded CE 6.0所支持的CPU。通过应用BSP可有效减少嵌入式设备的开发周期。

### 2. OAL

OAL（OEM Adaption Layer，原始设备制造商适配层）是位于Windows CE内核和目标硬件平台之间的一个代码层。OAL主要负责Windows CE与硬件通信，它与CPU、中断、内存、时钟和调试接口等核心设备相关，也是编译OAL各个模块代码后与其他内核库链接到一起形成Windows CE的内核可执行文档NK.exe的关键。简单来说，OAL用来屏蔽CPU平台的细节和抽象硬件功能，以实现操作系统内核的可移植性。

从逻辑结构上看，OAL位于Windows CE操作系统的内核与硬件之间，是连接Windows CE系统与硬件的枢纽；从功能上看，OAL类似桌面PC上的BIOS，具有初始化设备、引导操作系统以及抽象硬件功能等作用。但与BIOS不同的是，OAL隶属于操作系统，是操作系统的一部分。

### 3. KITL

KITL（Kernel Independent Transport Layer，内核独立传输层）为我们提供了一种调试Windows CE的简便方法。KITL将通信服务协议和用于通信的硬件分离开，所以我们在创建硬件传输层时就省去很多麻烦，否则我们自己必须实现与设备进行数据交互的协议。KITL工作在硬件传输层之上，因此，它无须关心用于通信的具体硬件，我们可以用USB、串行接口或者以太网接口作为KITL的调试通道，具体选择哪一个，由硬件平台和软件资源决定。有些设备没有以太网接口和串行接口，所以只能采用USB，如Mobile设备。如果系统采用了EBOOT，则建议使用以太网接口作为调试通道。这时，配置KITL的代价相对来说也较小。无论如何，KITL相当强大，在BSP的移植过程中，花一些时间来实现KITL的功能是完全值得的，基于KITL的实现，后续的调试过程中可以节省很多时间。

按启动顺序来说，KITL应该在OAL之后，内核之前启动。所以，必须先完成OAL的移

植，才能进一步移植KITL。

### 4. 内核

内核为Windows Embedded CE 6.0设备提供最基本的底层功能，这些功能包括服务程序、多线程（Multithread，指从软件或者硬件上实现多个任务并发执行的技术）、内存管理，还包括一些文件管理功能、程序和多线程的调度、实时支持、系统调用、内核电源管理等。NK.exe是OAL进程，内核Kernel.dll是伴随着OAL启动的，由此可知NK.exe是Windows Embedded CE 6.0的核心。图1-3表明了Windows Embedded CE 6.0的内核结构。

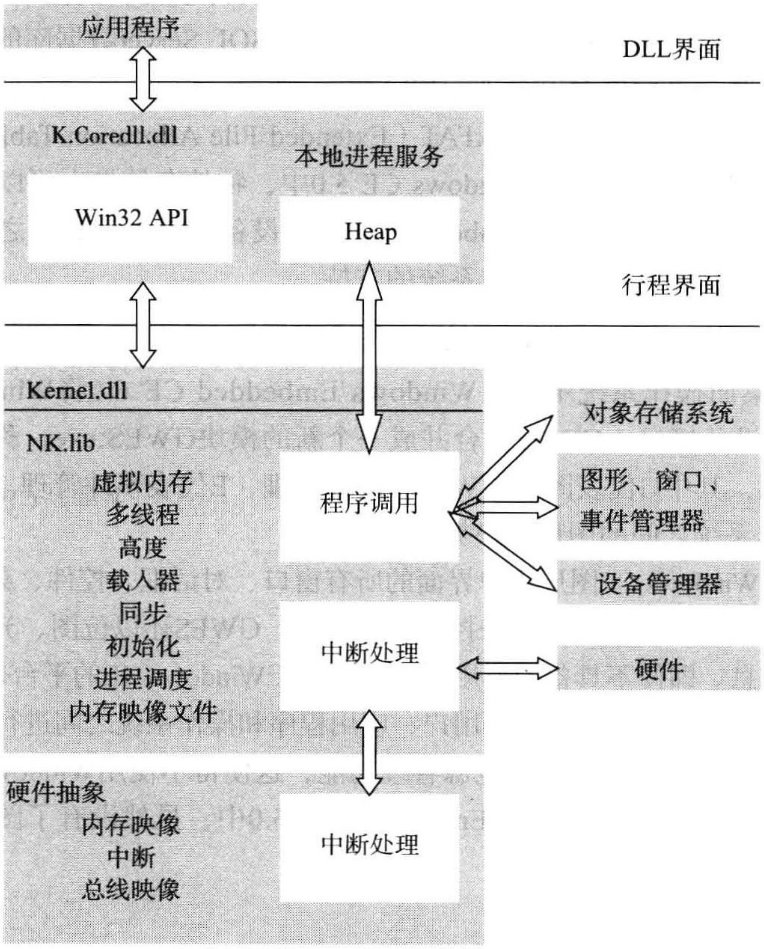



图1-3 Windows Embedded CE 6.0内核结构

图1-3中Kernel.dll的作用类似于Windows CE 5.0中的NK.exe。后面的章节将进一步对Kernel的程序调度、系统调用和存储系统方面进行介绍。

### 5. CoreDLL

CoreDLL同时出现在用户模式层和内核模式层中，它提供Core OS服务，使得应用程序能够访问Windows Embedded CE 6.0中的资源，如文件系统、内存、设备、程序和线程等。应用程

序也通过这些服务管理和监视其完成任务所需要的资源。通过CoreDLL，应用程序之间还可以共享程序和其他数据信息。

**6. 文件系统**

Windows CE文件系统采用了一种灵活的模块化设计，它允许自定义文件系统、筛选器和多种不同的块设备类型。文件系统和所有与文件相关的API都是通过Filesys.exe进程来管理的，Filesys.exe由ROM文件系统、存储管理器、对象存储三个组件组成。这个模块不但包括文件系统，还包括对存储对象的管理。Windows Embedded CE 6.0的对象存储系统有丰富的存储功能，提供对各种文件系统的支持，包括FAT文件系统、RAM文件系统和其他CD等通用磁盘文件系统。它还支持第三方定制的文件系统，支持与SQL-CE和SQL Server数据库的连接，同时拥有良好的安全特性，支持第三方程序对磁盘进行加密。

Windows Embedded CE 6.0采用了ExFAT（Extended File Allocation Table）。ExFAT不仅解决了大容量文件存储的限制问题（在Windows CE 5.0中，物件存储最大可以达到256MB，单个文件不超过4GB），还使得Windows Embedded CE 6.0设备与个人计算机之间的文件传输更方便、更容易。ExFAT还提供了对以前文件系统的支持。

**7. 图形、窗口和事件管理子系统**

与Microsoft的桌面操作系统不同，Windows Embedded CE 6.0将Win32 API的用户界面（USER32）和图形设备接口（GDI32）合并成一个新的模块GWES.exe，称为GWES子系统。GWES是一个缩写词，其中G代表图形，W代表窗口管理，E代表事件管理。GWES子系统是用户、应用程序和操作系统之间的图形用户界面。

GWES支持组成Windows CE图形用户界面的所有窗口、对话框、控件、菜单和资源，用户能够通过执行菜单命令、单机按钮等操作来控制应用程序。GWES还以位图、光标、文本以及图标等形式为用户提供信息。即使不具备图形用户界面的基于Windows CE的平台也可以使用GWES的基本窗口和消息功能，这些功能提供了在用户、应用程序和操作系统之间进行通信的方法。

Windows CE 5.0的GWES还包含了电源管理功能，这使得不使用Windows CE图形接口的设备无法使用电源管理功能，在Windows Embedded CE 6.0中，虽然没有了图形接口的嵌入式设备，但仍然可以使用电源管理功能。

**8. 设备管理器**

设备管理器是通过device.dll来实现的。Windows Embedded CE 6.0设备管理器（Device Manager，DM）的职能主要由三个模块完成。内核NK.exe加载device.dll，device.dll加载devmgr.dll，最后的工作主要由devmgr.dll来完成。DM负责设备驱动的加载、初始化以及卸载。DM将驱动程序的导出接口进行封装，对应用层提供系统API接口。DM可以在设备发生某特定事件时向应用程序发出通知。DM负责管理设备使用的内存和IO端口等资源。

**9. 驱动程序**

设备驱动程序是与硬件设备进行通信的系统程序。由于设备驱动程序的存在，大多数操作

系统上的应用程序都是与硬件无关的，应用程序的开发者和最终用户通常都不必关心底层的硬件到底是如何工作的。Windows CE操作系统之所以能够支持各种各样的硬件设备，毫无疑问，是因为对于每一款硬件设备来说都有其对应的设备驱动程序。

设备驱动程序是操作系统与硬件交互的方式，是连接硬件与操作系统的桥梁。通常，设备驱动程序在操作系统与硬件之间扮演着特殊的角色，它们是一个个独立的“黑盒子”，使某个硬件可响应一些定义良好的编程接口。这组标准化的接口通常由操作系统定义，与驱动程序的类型相关而与具体的硬件无关。

在Windows Embedded CE 6.0中，驱动程序有两种模式，一种是内核模式，另外一种是用户模式。在预设状况下，驱动程序执行在内核模式下，这有利于提高设备性能，但也增加了影响系统各方面性能的不确定因素，如果加入了不稳定的驱动到内核中，将会对嵌入式系统的可靠性、稳定性等多方面的性能产生致命的影响。所以驱动程序在发布和认证时，必须有严格的性能保证措施。

驱动程序根据各自类别的不同，将会被不同的程序加载，一般情况下，驱动程序会被文件系统（Filesys.dll）、设备管理器（device.dll）和GWES（GWES.dll）三种程序加载。

**10. 应用程序**

Windows Embedded CE 6.0应用程序的开发是Windows Embedded CE 6.0开发中很重要的一个环节。目前用得比较多的开发工具有EVC4（SP4）、VS2005、VS2008和Platform Builder。其中Platform Builder主要用来定制操作系统，虽然也支持编写应用程序，但相较于其他几种开发工具用得相对较少。VS2005和VS2008是微软的集成开发工具，Platform Builder 6.0就是作为VS2005的插件安装的。在VS2005中你可以编译Windows Embedded CE 6.0的操作系统，也可以编译设备上的应用程序，省去了切换开发环境的麻烦。Windows Embedded CE 6.0应用程序的开发相对桌面Windows应用程序的开发有如下这些特点：

- Unicode编码。Windows Embedded CE 6.0中的应用程序只能使用Unicode编码；桌面系统则支持Unicode和ANSI码。在移植PC端程序到设备上时需要注意这一点。
- SDK。SDK即软件开发支持包，Windows Embedded CE 6.0系统本身是一个非标准的操作系统，它的组件特性和可裁剪性决定了不同的系统支持的API是不同的，所以Windows Embedded CE 6.0中的SDK由系统开发人员在编译完系统后通过Platform Builder导出，应用程序开发人员安装此SDK并编写应用程序，最终将应用程序下载到目标平台上运行测试。

# 1.2 内存管理

## 1.2.1 内存结构

内存管理模型如图1-4所示，*号表示只在桌面Windows中存在。

1. 物理内存

物理内存是指在内部或外部总线上可访问的实际RAM/ROM。RAM分为对象存储区（Object Store，往往在关闭主电源后仍提供电力保证）和应用程序内存区（Program Memory）。ROM中存放的内容可以是压缩的，也可以是不压缩的（可本地执行XIP）。

Windows CE只能管理512MB的物理内存。

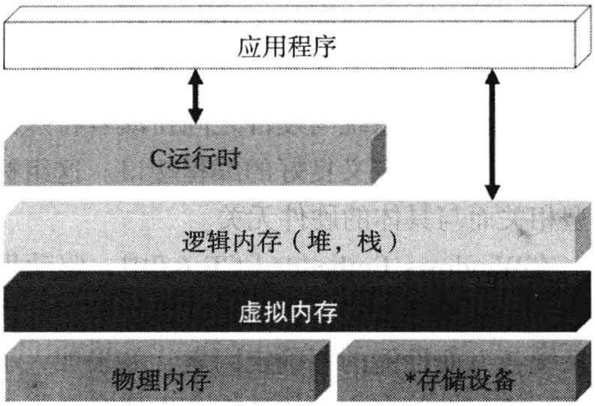


图1-4 内存管理模型

2. 虚拟内存

虚拟内存是通过内存管理单元（MMU）转换的虚拟地址空间。应用程序使用虚拟内存时不必关心实际物理内存，只要内存可用即可。操作系统使用MMU处理虚拟地址和物理地址间的实时转换，允许代码在需要的时候再换入。

Windows CE能管理4GB大小的虚拟内存。

3. 堆

堆是一段连续的、较大的虚拟地址空间，用来存放动态数据。创建进程时，会创建默认大小为192KB的堆。应用程序可创建自己的堆，在堆中可以动态地分配、释放所需大小的内存块。堆以字节为量度来分配内存，堆中的数据不可移动。堆的应用简化了虚拟内存分配，但易造成存储碎片。具有的特点：不可变更（当整个堆被释放时进行页面回收）、自动分配内存和按要求提交页。

4. 栈

栈一般用于存储一个函数中使用的临时数据，系统在每一个线程创建时为其分配默认的栈（64KB，栈顶2KB用来检测栈是否已满）。

5. 虚拟内存页面的三种状态

- free（可用）：页面可被分配。
- reserved（保留）：虚拟地址被保留，不能再分配，也不能被使用，因为它没有被映射到物理地址。
- committed（提交）：页面已经映射到物理内存。

### 1.2.2 Windows Embedded CE 6.0虚拟内存模型

与Windows CE 5.0类似，Windows Embedded CE 6.0系统的虚拟内存被划分为内核模式（Kernel Mode）层和用户模式（User Mode）层，两层各占2GB大小的空间。

内核模式层的虚拟内存空间分配如图1-5所示。

- 0x8000 0000 ~ 0xBFFF FFFF：静态虚拟地址。

- 0xC0000 000～0xC7FFF FFF：内核加载的（XIP）DLL。
- 0xC8000 000～0xCFFFF FFF：文件系统的对象存储区。
- 0xD0000 000～0xDFFFF FFF：内核模式的程序执行区，如GWES.DLL、系统DLL、内核驱动等。
- 0xE0000 000～0xEFFFF FFF：同上。除了具有SH4架构的CPU外。
- 0xF0000 000～0xFFFFF FFF：捕获系统调用，包含内核数据页。

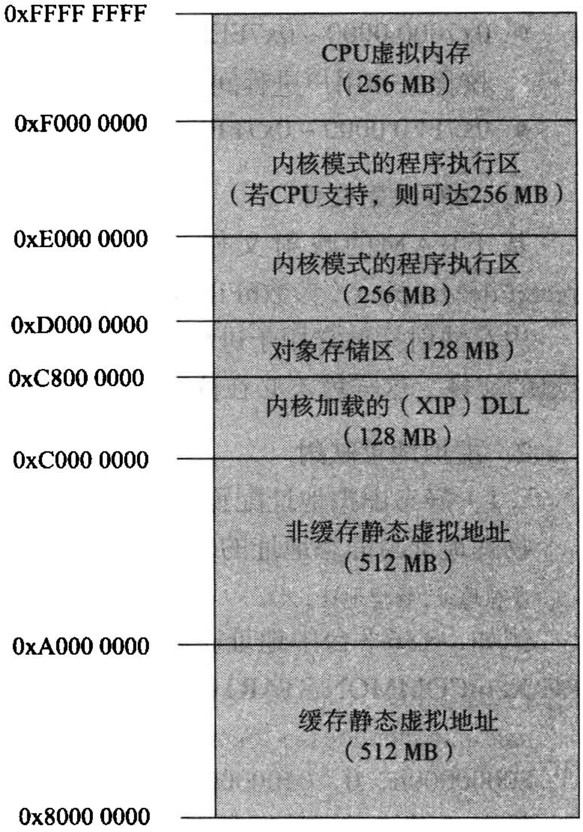



图1-5　内核模式层的地址空间分配

用户模式层的虚拟地址空间被一分为二。最低的64KB被系统保留，用来捕获野指针。低1GB的地址空间用来加载进程的代码、数据、堆、栈等。具体分配如图1-6所示。

- 0x00000 000～0x0000F FFF（64KB）：系统保留。
- 0x00010 000～0x3FFFF FFF（1GB）：用来加载进程的代码、数据、堆、栈等。
- 0x40000 000～0x5FFFF FFF（512MB）：加载正在运行的不同进程的DLL代码和只读数据。空间分配从低向高。

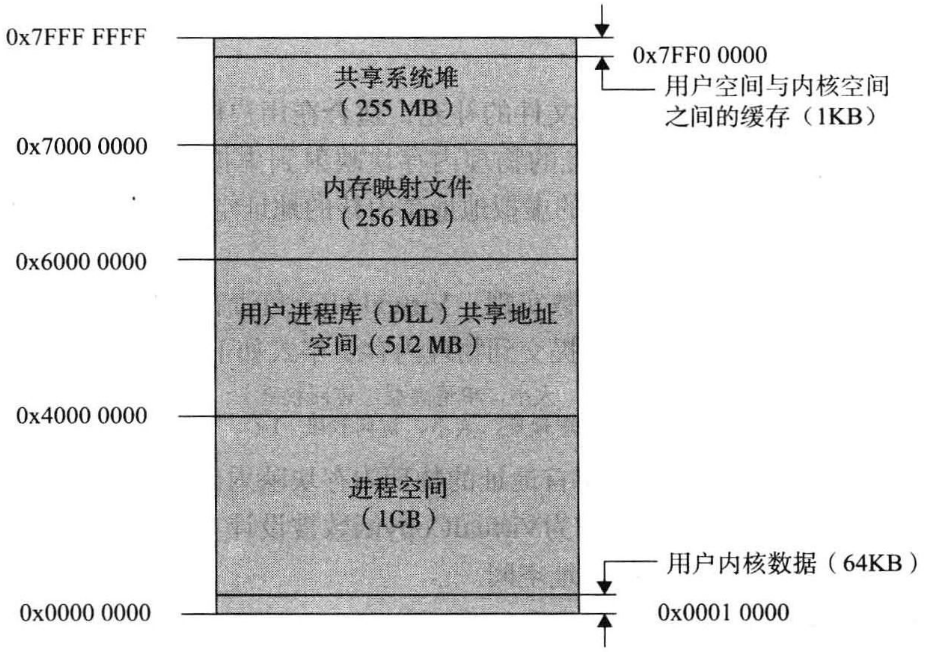



图1-6　user mode（用户模式）下的地址空间分配

- 0x6000 0000 ~ 0x6FFF FFFF（256MB）：分配给内存映射文件（Memory-Mapped File，MMF），后面会详细介绍。
- 0x7000 0000 ~ 0x7FEF FFFF（255MB）：内核程序和用户进程间的共享系统堆，实现内核程序和用户进程间通信（无须通过API）。此区域用户程序只读，但操作系统可写。
- 0x7FF0 0000 ~ 0x7FFF FFFF（1MB）：不可访问，作为用户和内核间的缓存。

### 1. 内存映射文件

基于RAM的映射文件为内存映射文件对象，其不以实际文件为基础。通过调用CreateFileMapping（参数hFile设置为INVALID_HANDLE_VALUE）可获得这些文件。

内存映射文件常用于进程间通信。另外，它可以是加载的各类文件。还可使用它来映射大容量的文件，这样就不必在读取文件数据前设置很大的缓冲区。

### 2. 虚拟地址映射

（1）静态虚拟地址配置文件

物理地址与静态地址的映射关系是在一个名为OEMAddressTable的表中定义的。格式为：

```
虚拟地址,物理地址,大小
```

例如，x86平台中地址映射在startup.asm中，目录是WINCE600\PLATFORM\COMMON\SRC\X86\COMMON\STARTUP，如：

```
RAM 0x80000000 -> 0x00000000,size 64M
```

80000000h，0，04000000h表示将虚拟地址80000000映射到物理地址0，大小为64MB。

静态映射的虚拟地址空间一部分只能由内核访问，还可通过调用API来实现，函数声明如下所示：

```
LPVOID CreateStaticMapping(
   DWORD dwPhysBase,
   DWORD dwSize
);
```

CreateStaticMapping函数作为配置文件的补充，适合在用户模式的应用程序或驱动程序中调用，调用这个函数能够将指定首地址的物理内存块映射到虚拟地址空间，函数返回虚拟地址。范围在C400 0000 ~ E000 0000之间的虚拟地址是内核的地址空间，只能由内核访问。

（2）动态虚拟地址映射

动态虚拟地址映射通过两个API函数实现：VirtualAlloc负责在虚拟内存空间内保留一段虚拟内存，而VirtualCopy负责将虚拟内存提交到物理内存。格式如下：

```
LPVOID VirtualAlloc(虚拟起始地址，大小，申请类型，访问权限);
BOOL VirtualCopy(虚拟目标地址，物理地址，大小，访问权限 );
```

VirtualCopy函数用于动态地将指定首地址的物理内存块映射到虚拟地址空间，这个虚拟地址空间一般为用户进程的地址空间，因为VirtualCopy函数被设计专用于驱动程序调用，常用于将一个物理内存块映射到不同的虚拟地址空间。

**分配和保留虚拟内存。**

VirtualAlloc的函数声明如下所示：

```
LPVOID VirtualAlloc(
LPVOID lpAddress,
DWORD dwSize,
DWORD flAllocationType,
DWORD flProtect);
```

- 参数lpAddress是要分配内存区域的地址。
- 参数dwSize是要分配或者保留的区域的大小。
- 参数flAllocationType指定了分配的类型，其可以指定或者合并以下标志：MEM_COMMIT、MEM_AUTO_COMMIT、MEM_RESERVE和MEM_TOP_DOWN。其中MEM_COMMIT标志用于分配程序使用的内存，MEM_RESERVE用于保留虚拟地址空间以便以后提交。
- 参数flProtect指定了被分配区域的访问保护方式，参数取值如下所示：
- PAGE_READONLY，表示该区域为只读。如果应用程序试图访问区域中的页的时候，将会被拒绝访问。
- PAGE_READWRITE，表示区域可被应用程序读写。
- PAGE_EXECUTE，表示区域包含可被系统执行的代码。试图读写该区域的操作将被拒绝。
- PAGE_EXECUTE_READ，表示区域包含可执行代码，应用程序可以读该区域。
- PAGE_EXECUTE_READWRITE，表示区域包含可执行代码，应用程序可以读写该区域。
- PAGE_GUARD，表示区域第一次被访问时进入一个STATUS_GUARD_PAGE异常，这个标志要与其他保护标志合并使用，表明区域被第一次访问的权限。
- PAGE_NOACCESS，表示任何访问该区域的操作都将被拒绝。
- PAGE_NOCACHE，表示RAM中的页映射到该区域时将不会被微处理器缓存。

**释放虚拟内存。**

通过调用VirtualFree来取消提交或释放虚拟内存，即从物理RAM页中取消提交或者取消映射，但是保持页被保留的状态。格式如下：

```
BOOL VirtualFree(
  LPVOID lpAddress,
  DWORD dwSize,
  DWORD dwFreeType);
```

- 参数lpAddress是一个指针，指向要被释放或取消提交的虚拟内存的区域。
- 参数dwSize指明要取消提交区域的大小，以字节为单位。如果区域要被释放，这个值必须是0。
- 参数dwFreeType包含了操作类型标志：MEM_DECOMMIT标志指定了区域将被取消提交但是仍保留；MEM_RELEASE标志说明区域要取消提交并且释放。虚拟内存管理函数如表1-1所示。

**表1-1 虚拟内存管理函数**

| 函 数 | 功 能 |
|---|---|
| VirtualAlloc | 保留虚拟内存 |
| VirtualQuery | 提供虚拟内存页的状态信息 |
| VirtualProtect | 改变提交页面的访问权限 |
| VirtualFree | 取消提交或释放虚拟内存 |
| VirtualCopy | 绑定指定的物理内存地址范围到静态虚拟地址 |

（3）举例

通过函数VirtualAlloc和VirtualCopy，把芯片中I/O端口的物理地址和操作系统的虚拟内存空间联系起来。其中sz是申请的长度，addr是申请虚拟地址空间的实际物理端口寄存器地址在Windows CE中的映射地址。

```
/*地址为0，长度为sz，保留虚拟地址空间以便以后提交，任何访问该区域的操作将被拒绝。*/
  reg = (PVOID)VirtualAlloc(0,sz,MEM_RESERVE,PAGE_NOACCESS) ;
  if(reg) {
/*虚拟目标地址reg,物理地址addr,大小sz,区域可被应用程序读写,RAM中的页映射到该区域时将不会被微处理器缓存。*/
if( ! VirtualCopy(reg,addr,sz,PAGE_READWRITE | PAGE_NOCACHE) ) {
VirtualFree(reg,sz,MEM_RELEASE) ;}
}
```

# 1.3 文件系统与存储管理

## 1.3.1 文件系统

### 1. 文件系统类型

文件系统提供对存储设备的高效访问，并将存储设备结构化以便读写。

Filesys将所有文件系统统一到一个根“\”下面的单系统中，系统中无驱动器号，驱动器作为文件夹装在根的下面，其路径类似于“\Storage Card”。

Windows Embedded CE 6.0提供三种文件系统：RAM文件系统、ROM文件系统和可安装文件系统。

- RAM文件系统：通常连接到呈现给应用程序的统一文件系统的根。Windows Embedded CE 6.0中使用的文件系统的根目录为“\”，所有的文件系统都被统一到根目录“\”下面，ROM文件系统通常直接挂载到“\Windows”目录，RAM文件系统通常直接挂载到根目录下，也就是说，根目录下除了挂载的外存目录和“\Windows”目录之外的所有文件都位于RAM文件系统中。
- ROM文件系统：连接到统一文件系统中的“\Windows”文件夹。Windows Embedded CE 6.0的内部文件系统控制对ROM的访问，也提供了对位于RAM的对象存储的文件存储，Windows Embedded CE 6.0支持两种定制的文件系统：RAM and ROM文件系统和ROM文件系统。RAM and ROM文件系统提供了对象存储里的文件存储以及对ROM的访问，而ROM文件系统不允许应用程序将文件存储在对象存储里，需要通过根目录下的Windows目录才可以访问ROM中的数据。ROM文件系统连接到统一文件系统中的“\Windows”文件夹，这意味着ROM中的所有文件均可作为“\Windows”文件夹中的只读文件来访问。
- 可安装文件系统：外设可作为用户的文件系统挂载，如CD驱动器常挂载到“\CD Driver”目录。也可根据设备需要自定义文件系统。

### 2. 文件系统格式

- FAT

FAT（File Allocation Table）适用于所有外部存储卡，包括ATA卡（IDE接口）、线性FLASH卡以及PC卡等，卡可被分区，每个分区可作为一个FAT卷挂载，放置在根目录的一个文件夹下。

- BinFS

BinFS（Binary ROM Image File System）用于读取由Romimage.exe产生的二进制文件格式，其中.bin二进制文件格式将数据按特定的分段组织。

- CDFS/UDFS

CDFS（Compact Disc File System）和UDFS（Universal Disc File System） 用于读取CD、DVD和CD-ROM。

### 3. 操作文件的API

- 创建并打开文件：CreateFile
- 读文件：ReadFile
- 写文件：WriteFile
- 移动文件指针：SetFilePointer
- 关闭文件：CloseHandle
- 获取文件信息：GetFileAttributes
- 获取文件时间：GetFileTime
- 获取文件大小：GetFileSize

（1）创建和打开文件

创建文件或者打开已存在的文件或设备可以使用以下标准的Win32函数来完成：

```
HANDLE CreateFile(LPCTSTR lpFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
DWORD dwCreationDistribution,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile);
```

- 参数lpFileName是要打开或创建的文件的文件名。文件的名称应该包括完全指定的路径。对于没有路径信息的文件名，系统默认为在对象存储库的根目录下。
- 参数dwDesiredAccess指出了所需的访问权限。允许使用的标志是用来请求对文件进行读取访问的GENERIC_READ和对文件进行写入访问的GENERIC_WRITE。
- 参数dwShareMode指定了可以给予其他进程访问的权限。这个参数取值可以是FILE_SHARE_READ或FILE_SHARE_WRITE。
- 参数lpSecurityAttributes设置为NULL时Windows CE将忽略安全性。
- 参数dwCreationDistribution能告诉CreateFile怎样打开或创建文件，该参数的取值如下所示：
- CREATE_NEW：创建一个新文件，如果该文件已经存在，则该函数将失败。

- CREATE_ALWAYS：创建一个新文件或覆盖一个现有的文件。
- 参数dwFlagsAndAttributes将定义该文件的属性标志。

（2）读取和写入

Windows Embedded CE 6.0支持标准的Win32函数ReadFile和WriteFile，完成对文件的读取操作可以调用如下函数，代码如下所示：

```
BOOL ReadFile(
HANDLE hFile,
LPVOID lpBuffer,
DWORD nNumberOfBytesToRead,
LPDWORD lpNumberOfBytesRead,     LPOVERLAPPED lpOverlapped);
```

- 参数hFile是要读取的已打开文件的句柄。
- 参数lpBuffer是指向将接收数据的缓冲区的指针。
- 参数nNumberOfBytesToRead是要从文件中读取的字节数。
- 参数lpNumberOfBytesRead是指向接收实际读取的字节数的指针。
- 参数lpOverlapped必须设置为NULL。

完成对文件的写入操作可以调用下面这个函数，代码如下所示：

```
BOOL WriteFile(
HANDLE hFile,
LPCVOID lpBuffer,
DWORD nNumberOfBytesToWrite,
LPDWORD lpNumberOfBytesWritten,
LPOVERLAPPED lpOverlapped);
```

该函数的参数与ReadFile函数中的参数基本相同，只是参数lpBufter现在指向的是要写入文件的数据，数据被写入到由文件指针指出的文件偏移量处，当写入操作完成之后会更新该文件指针，因此它指向的字节会刚好在被写入的数据后面。同Read File函数一样，lpOverlapped参数必须为NULL。

## 1.3.2 注册表

### 1. 注册表的作用

注册表保存着应用程序、驱动、用户参数配置和其他配置设定的数据。

Windows Embedded CE 6.0系统中的注册表和Windows操作系统中注册表的概念和结构基本相同，主要分为根键、子键、键值项。例如：

```
[ROOT]\[Key]\[Subkey]\[Subkey or entry]
```

Windows Embedded CE 6.0系统中的注册表支持四个根键，描述如表1-2所示。

表1-2 键名及描述

| 键　名 | 描　述 |
|---|---|
| HKEY_LOCAL_MACHINE | 硬件和驱动配置数据 |
| HKEY_CURRENT_USER | 用户配置数据 |
| HKEY_CLASSES_ROOT | OLE和文件类型匹配配置数据 |
| HKEY_USERS | 适用于所有用户的数据 |

键值项由三部分组成：名称、类型、数据。

键值项类型主要包括如下几种：

- REG_BINARY：二进制数据。

- REG_DWORD：32位双字数据。
- REG_SZ：以0结尾的字符串。
- REG_DWORD_BIG_ENDIAN：双字big-endian格式，一个多字节值在内存中被从高字节到低字节分段存储。例如，值0x12345678在big-endian格式中被存储为（0x12 0x34 0x56 0x78）。
- REG_DWORD_LITTLE_ENDIAN：一个little-endian格式的32位数字。little-endian格式是一个多字节值在内存中被从低字节到高字节分段存储，存储过程正好与big-endian格式相反。例如，值0x12345678在little-endian格式中被存储为（0x78 0x56 0x34 0x12）。
- REG_EXPAND_SZ：扩展字符串。可以是包含未被引用的环境变量字符串（例如，"%PATH%"）。
- REG_LINK：一个Unicode符号链接。
- REG_RESOURCE_LIST：设备驱动程序资源列表。
- REG_MULTI_SZ：多个字符串（字符串数组）。
- REG_NONE：未定义值类型。

### 2. 注册表类型

Windows Embedded CE 6.0支持两种类型的注册表：一种基于RAM，一种基于配置单元（hive）。

（1）基于RAM的注册表

Windows Embedded CE 6.0把注册表作为一个对象存放在对象存储中，在对象存储中保存所有注册表数据。设备必须为RAM持续供电，设备断电则丢失数据（不为RAM供电的设备必须在关闭电源过程中备份注册表，在恢复供电时还原注册表）。基于RAM的注册表用于从不或很少冷启动的设备。

Windows Embedded CE 6.0提供了两种方法用来断电保存基于RAM的注册表。

第一种方法：Windows Embedded CE 6.0提供了两个系统API函数用来保存和还原整个注册表，它们的声明如下：

```
BOOL RegCopyFile(LPCWSTR lpszFile);
BOOL RegRestoreFile(LPCWSTR lpszFile);
```

lpszFile：保存注册表信息的文件的名字。

第二种方法：OEM可在BSP的OAL层中实现两个函数，用来保存和还原整个注册表，它们的声明如下：

```
DWORD ReadRegistryFromOEM(DWORD dwFlags,
LPBYTE lpData,DWORD cbData);
BOOL WriteRegistryToOEM(DWORD  dwFlags,
LPBYTE lpData,DWORD cbData);
```

- dwFlags：REG_READ_BYTES_START表示读取新的注册表，REG_WRITE_BYTES_START表示写入新的注册表。
- lpData：指向注册表数据的缓冲区，由操作系统分配。

- cbData：缓冲区的大小，0表示到达注册表尾部。

例如，下例使用基于RAM的注册表，利用RegCopyFile和RegRestoreFile实现永久保存注册表数据。

```
CString  strRegBackup = L"\\hard disk\\RegBackup.reg";
//导出。在系统关闭前。
if(! RegCopyFile(strRegBackup))          //导出注册表到RegBackup.reg.
return FALSE;
//导入。在系统启动时。
if(! RegRestoreFile(strRegBackup))       //恢复注册表
return  FALSE;
//重新热启动,只有启动时,才会检测RAM中的注册表
if(! KernelIoControl(IOCTL_HAL_REBOOT,NULL,0,NULL,0,NULL))
return  FALSE;
```

（2）基于配置单元（hive）的注册表

Windows Embedded CE 6.0把注册表数据存放在文件系统的文件或配置单元中，配置单元是注册表中一组注册表项、子项、值的集合，一个配置单元作为一个文件进行保存和还原。

在文件或配置单元中存储数据，方便在任何文件系统中保存，并且减少了系统冷启动的负担。

基于hive的注册表用在经常需要冷启动的设备上非常高效，而很少或者从不用在热启动的设备上，最适合用在有永久保存介质或者多用户的设备上。基于hive的注册表也提供不同的用户hive，所以可以为每一个用户定义不同的注册表配置信息，这样，一个多用户的系统会包含多个用户的hive（user.hv），一个用户的hive在登录时被加载，注销的时候被卸载。

基于hive的注册表包含三部分：引导hive、系统hive和用户hive，分别对应于boot.hv、system.hv和user.hv。boot.hv存放系统引导时所需的注册表设置，这部分设置不会保存在FLASH或者磁盘上面，也就是说，在系统掉电以后就会丢失。实际上，这部分应该说是属于RAM-Based注册表，由于这部分数据不需要改动，所以也就无所谓了。system.hv存放关于系统的注册表设置，user.hv存放和用户相关的注册表设置。

### 1.3.3 存储管理器

存储管理器负责管理系统中所有的外围存储设备，包括管理它们所有的文件系统和存储驱动。所有的文件、数据和容量都是通过存储管理器来操作的，存储管理器的功能由系统中的fsdmgr.dll模块实现，包括如下三部分：

- 文件系统驱动管理器（File System Driver Manager，FSD Manager）
- 分区管理器（Partition Manager）
- 存储驱动管理器（Block Driver Manager）

存储管理器的体系结构如图1-7所示。

从图1-7可以知道存储管理器是一个分层结构，由文件系统过滤器（file system filter）、文件系统驱动（file system driver）、分区驱动（partition driver）和存储驱动（block driver）组成。

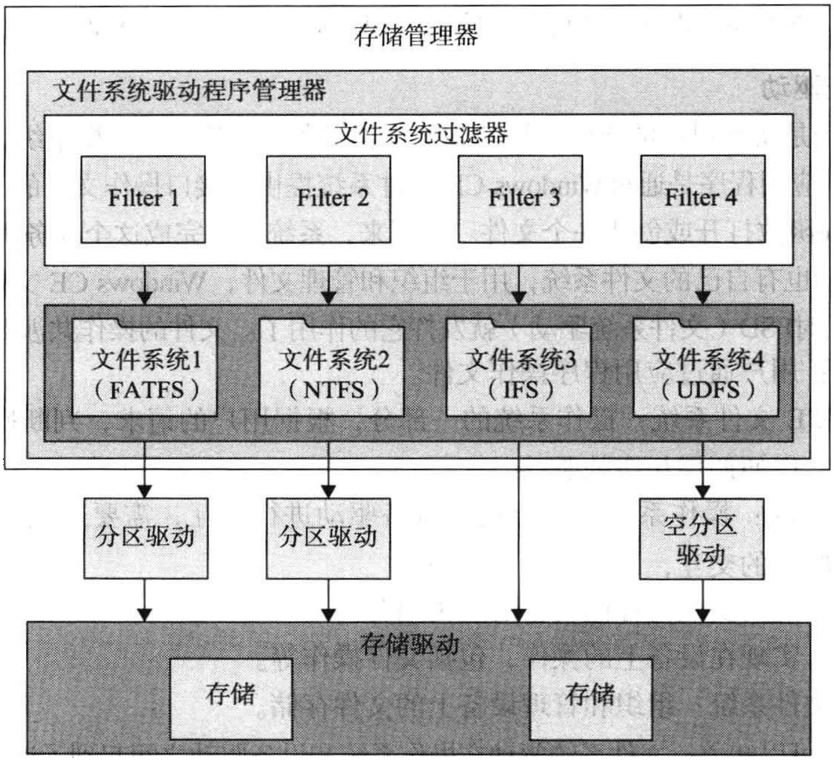



图1-7　存储管理器的体系结构

### 1. 文件系统过滤器

文件系统过滤器是一个导出文件系统入口的DLL，这些入口映射到标准的文件系统函数，比如CreateFile函数和CreateDirectory函数。文件系统过滤器用于处理对文件系统的调用，之后文件系统才能获得这些调用，也就是说存储管理器在调用文件系统API之前先调用文件系统过滤器的过滤函数，这样就允许对文件访问进行某些特殊的处理，因为文件系统过滤器拦截对文件系统的调用，所以可以对文件数据进行加密、解密、压缩甚至扫描文件是否存在病毒等。

文件系统过滤器是轻量级（lightweight）的文件系统驱动（FSD），它导出的钩子函数（hook function）是FSD_HookVolume()和FSD_UnhookVolume()，而不是磁盘挂载函数FSD_MountDisk()和FSD_UnmountDisk()，下面分别介绍这几个函数。

- FSD_HookVolume()：在存储管理器加载文件系统过滤器的时候会调用这个入口函数，此函数是由文件系统导出的函数，并且被文件系统磁盘管理器（File System Disk Manager，FSDMGR）直接调用。
- FSD_UnhookVolume()：在存储管理器卸载文件系统过滤器的时候会调用这个入口函数，此函数是由文件系统导出的函数，并且被文件系统磁盘管理器（FSDMGR）直接调用。
- FSD_MountDisk()：设备管理器调用这个函数来挂载包含具体文件类型的指定磁盘，比如可以在根目录下挂载ResidentFlash盘符，这个盘符是基于FATFS来操作nandflash的。
- FSD_UnmountDisk()：设备管理器调用这个函数来卸载指定的盘符。

文件系统驱动管理器负责加载文件系统驱动（Files System Drivers，FSD）和文件系统过滤器。

### 2. 文件系统驱动

文件系统驱动是实现对应的文件系统的DLL，它将存储设备上的数据组织为文件和文件夹来对设备进行操作。应用程序是通过Windows CE 文件系统提供的接口操作文件的，例如，应用程序可以通过CreateFile函数打开或创建一个文件。接下来，系统如何完成这个任务呢？比如操作SD卡上的文件，SD卡上也有自己的文件系统，用于组织和管理文件，Windows CE 文件系统是如何操作这些文件的呢？此时FSD（文件系统驱动）就发挥它的作用了。文件的操作共涉及六个层次：

- 应用程序：用户通过应用程序操作文件。
- Windows CE 文件系统：操作系统的一部分，根据用户的请求，判断操作的是哪个设备上的文件，以便调用相应的驱动。
- 文件系统驱动：操作系统无法直接与设备驱动进行交互，需要通过文件系统驱动，实现与设备驱动的交互，以实现文件操作。
- 分区驱动：主要实现对存储设备的逻辑划分。
- 设备驱动：实现在设备上的操作，包括文件操作等。
- 设备上的文件系统：组织和管理设备上的文件存储。

从以上的分析可以知道，文件系统驱动在操作系统和设备驱动之间起到了连接作用。在实现的时候，文件系统驱动基于设备驱动，实现了Windows CE 文件系统中的文件操作函数，如CreateFile等。

### 3. 分区驱动

分区是对存储设备的逻辑划分，Windows Embedded CE 6.0允许物理磁盘（常用的是nandflash）包含多个分区，并且每个分区可以格式化为不同的文件系统，但是每个设备只能由一个分区驱动程序来管理。下面是在src/bootloader/eboot/nand.cpp文件的WriteOSImageToBootMedia函数中关于建立分区的代码：

```
//加1是因为MBR存储在第IMAGE_START_BLOCK个block中
hPart=BP_OpenPartition((IMAGE_START_BLOCK+1)*PAGES_PER_BLOCK,
SECTOR_TO_BLOCK_SIZE(FILE_TO_SECTOR_SIZE(dwBINFSPartLength))*PAGES_PER_BLOCK,
PART_BINFS,//创建BINFS格式的文件系统
TRUE,//活动分区
PART_OPEN_ALWAYS)/*如果不存在这个分区则创建,如果已存在则打开这个分区*/
if(hPart==INVALID_HANDLE_VALUE)
{
 OALMSG(OAL_ERROR,(TEXT("ERROR;kandi WriteOSImageToBootMedia:Failed to open/
     create partition.\r\n")));
Return(FALSE);
}
```

上述代码主要是通过分区驱动在nandflash上建立一个使用BINFS的分区，这个分区用于保存NK映像。

```
hPart=BP_OpenPartition(NEXT_FREE_LOC,
USE_REMAINING_SPACE, //指示FLASH中剩余的空间被用于FAT分区,也只有在创建一个扩展分区的时候才
能用这个值
PART_DOS32,//FAT32
```

```
TRUE,
PART_OPEN_ALWAYS);
if(hPart==INVALID_HANDLE_VALUE)
{
OALMSG(OAL_WARN,(TEXT("WARN:sTOREImageToBootMedia:Failed to open/create Extended
    partition.\r\n")));
Return(FALSE);
}
```

上述代码主要是通过分区驱动在nandflash上建立一个使用FAT32文件系统的分区，这个分区建立在除去NK映像占用的其余nandflash空间上。

分区驱动实际上是存储驱动的转换器。它公开与存储驱动相同的接口，并将分区的块地址（逻辑地址）转换为存储设备块的真实地址（物理地址）。然后，它将调用传递给存储驱动。分区驱动是一个DLL（mspart.dll），在common.reg中的相关注册表信息如下所示：

```
; HIVE BOOT SECTION
; Partition driver key.
[HKEY_LOCAL_MACHINE/System/StorageManager/MSPART]
"Dll"="mspart.dll"
; END HIVE BOOT SECTION
```

#### 4. 存储驱动

存储驱动用于和存储设备直接打交道，可以读写存储设备。存储驱动也称为“块驱动”，因为它们提供对数据存储的随机寻址块的访问。对于存储设备为Nand Flash的系统，其存储驱动就是Nand Flash驱动。

## 1.4 进程与线程

### 1.4.1 Windows CE进程

进程是应用程序的执行实例，每个进程都是由私有的虚拟地址空间、代码、数据和其他各种系统资源组成。进程在运行过程中创建的资源随着进程的终止而销毁，所使用的系统资源在进程终止时被释放或关闭。进程是一个具有一定独立功能的程序在一个数据集合上一次动态执行的过程。进程包含程序、数据、PCB三部分。

下面简单介绍系统进程、与进程相关的API以及如何创建进程。

#### 1. 系统进程

- NK.exe：系统服务
- Filesys.dll：对象存储
- GEWS.dll：图形、窗口和事件服务
- device.dll：设备管理
- explorer.dll：shell
- services.exe：服务管理
- repllog.exe：系统与外部设备连接

- rapisrv.exe：系统与Windows连接

## 2. 与进程相关的API函数

（1）创建进程

创建进程用CreateProcess函数。

（2）结束进程

- main或WinMain调用返回。
- 调用ExitProcess函数或主线程调用ExitThread函数。
- 调用TerminateProcess函数。

（3）结束进程时，Windows CE的操作

- 终止该进程内所有线程。
- 释放用户对象和GDI对象。
- 关闭所有内核对象，进程退出。
- 内核对象计数器减1。

（4）获取进程终止状态

GetExitCodeProcess函数。

创建进程的API函数定义如下所示：

```
BOOL CreateProcess(
LPCWSTR    lpApplicationName,      //可执行文件的路径和名字,不能为NULL
LPWSTR       LpCmdLine,            //传递启动参数(必须为Unicode字符串)
LPSECURITY_ATTRIBUTES   lpProcessAttributes,     //不支持,设置为NULL
LPSECURITY_ATTRIBUTES   lpThreadAttributes,      //不支持,设置为NULL
BOOL              bInheritHandles,               //不支持,设置为FALSE
DWORD            dwCreationFlags,                //进程加载后的初始状态
PVOID            lpEnvironment,                  //不支持,设置为NULL
LPCWSTR          lpCurrentDirectory,             //不支持,设置为NULL
LPSTARTUPINFOW              lpStartupInfo,       //不支持,设置为NULL
LPPROCESS_INFORMATION  lpProcessInformation,     //返回的进程相关的信息
);
```

整个函数返回值为BOOL型，当成功创建进程后，返回值为真（TRUE），否则返回假（FALSE）。

## 3. 创建进程

```
TCHAR szFileName[MAX_PATH];
TCHAR szCmdLine[64];
DWORD dwCreationFlags;
PROCESS_INFORMATION pi;
int rc;
lstrcpy(szFileName,TEXT("calc"));
lstrcpy(szCmdLine,TEXT(""));
dwCreationFlags = 0; //创建一个标准的进程
rc = CreateProcess(szFileName,szCmdLine,NULL,NULL,FALSE,
dwCreationFlags,NULL,NULL,NULL, &pi);
if(rc)
```

```
{ MessageBox(hWnd,L"创建进程成功！ ",TEXT("ffh提示"),MB_OK);
  CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);
}
```

上述代码将启动手持PC中的标准“计算器”小程序。因为没有给文件名指定路径，所以CreateProcess函数将使用标准的Windows CE搜索路径在\WindowsCE目录下查找calc.exe。因为没有将命令行传递给calc，所以应该将NULL传递给形参LpCmdLine，将实参szCmdLine设置为NULL，以便将LpCmdLine参数与CreateProcess函数中没有使用的其他参数区别开，而对于形参dwCreationFlags也做了同样的处理。如果调用CreateProcess成功，它将返回非零值，上面这段代码能对是否成功创建进程进行检测，并且如果调用成功就关闭在PROCESS_INFORMATION结构中返回的进程和线程句柄。所有的Win32应用程序都必须这样做，以防止内存耗尽。

## 1.4.2 Windows CE线程

线程是进程内部的一个执行单元。系统创建好进程后，实际上就启动执行了该进程的主执行线程，主执行线程采用函数地址形式，比如说main或WinMain函数将程序的启动点提供给Windows CE系统。主执行线程终止了，进程也随之终止。线程是进程中的一个实体，是被系统独立调度和分派的基本单位。

线程只拥有很少的在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。

每一个进程至少有一个主执行线程，它无需用户主动创建，是由系统自动创建的。用户根据需要在应用程序中创建其他线程，多个线程并发地运行于同一个进程中。一个进程中的所有线程都在该进程的虚拟地址空间中，共同使用这些虚拟地址空间、全局变量和系统资源。

一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。

### 1. 与线程相关的API函数

Win32提供了一系列的API函数来完成线程的创建、挂起、恢复、终结以及通信等工作。下面将选取其中的一些重要函数进行说明。

（1）创建一个新的线程

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId);
```

该函数在其调用进程的进程空间里创建一个新的线程，并返回已建线程的句柄，其中各参数说明如下：

- lpThreadAttributes：指向一个SECURITY_ATTRIBUTES结构的指针，该结构决定了线程的安全属性，一般设置为NULL。
- dwStackSize：指定了线程的堆栈深度，一般都设置为0。
- lpStartAddress：表示新线程开始执行时代码所在函数的地址，即线程的起始地址。一般

情况为(LPTHREAD_START_ROUTINE)ThreadFunc，ThreadFunc是线程函数名。

- lpParameter：指定了线程执行时传送给线程的32位参数，即线程函数的参数。
- dwCreationFlags：控制线程创建的附加标志，可以取两种值。如果该参数为0，线程在被创建后就会立即开始执行；如果该参数为CREATE_SUSPENDED，则系统产生线程后，该线程处于挂起状态，并不马上执行，直至函数ResumeThread被调用。
- lpThreadId：该参数返回所创建线程的ID。

如果创建成功则返回线程的句柄，否则返回NULL。

（2）挂起指定的线程

```
DWORD SuspendThread(HANDLE hThread);
```

该函数用于挂起指定的线程，如果函数执行成功，则终止执行线程。

（3）结束线程的挂起状态

```
DWORD ResumeThread(HANDLE hThread);
```

该函数用于结束线程的挂起状态，执行线程。

（4）线程终结自身的执行

```
VOID ExitThread(DWORD dwExitCode);
```

该函数用于线程终结自身的执行，主要在线程的执行函数中调用。其中参数dwExitCode用来设置线程的退出码。

（5）强行终止某一线程

```
BOOL TerminateThread(HANDLE hThread,DWORD dwExitCode);
```

一般情况下，线程运行结束之后，线程函数正常返回，但是应用程序可以调用TerminateThread强行终止某一线程的执行。各参数含义如下：

- hThread：将被终结的线程的句柄。
- dwExitCode：用于指定线程的退出码。

使用TerminateThread函数终止某个线程的执行是不安全的，可能会引起系统不稳定。虽然该函数立即终止线程的执行，但并不释放线程所占用的资源。因此，一般不建议使用该函数。

### 2. 创建线程举例

建立一个基于对话框的工程MultiThread1，在对话框IDD_MULTITHREAD1_DIALOG中加入两个按钮和一个编辑框，两个按钮的ID分别是IDC_START和IDC_STOP，标题分别为“启动”和“停止”，IDC_STOP的属性选中Disabled；编辑框的ID为IDC_TIME，属性选中Read-only。

在MultiThread1Dlg.h文件中添加线程函数声明：void ThreadFunc()。

注意，线程函数的声明应在类CMultiThread1Dlg的外部。在类CMultiThread1Dlg内部添加protected型变量：HANDLE hThread和DWORD ThreadID，分别代表线程的句柄和ID。

在MultiThread1Dlg.cpp文件中添加全局变量m_bRun：volatile BOOL m_bRun，m_bRun代表线程是否正在运行。要注意全局变量m_bRun是使用volatile修饰符的，volatile修饰符的作用是告诉编译器无需对该变量做任何优化，即无需将它放到一个寄存器中，并且该值可被外部改

变。对于多线程引用的全局变量来说，volatile是一个非常重要的修饰符。

线程函数的代码如下：

```
void ThreadFunc()
{
CTime time;
CString strTime;
m_bRun=TRUE;
while(m_bRun)
{
time=CTime::GetCurrentTime();
strTime=time.Format("%H:%M:%S");
::SetDlgItemText(AfxGetMainWnd()->m_hWnd,IDC_TIME,strTime);
Sleep(1000);
}
}
```

该线程函数没有参数，也不返回函数值。只要m_bRun为TRUE，线程一直运行。

双击IDC_START按钮，完成该按钮的消息函数：

```
void CMultiThread1Dlg::OnStart()
{
  // TODO: 在此处添加控制通知句柄代码
  hThread=CreateThread(NULL,
  0,
  (LPTHREAD_START_ROUTINE)ThreadFunc,
  NULL,
  0,
  &ThreadID);
  GetDlgItem(IDC_START)->EnableWindow(FALSE);
  GetDlgItem(IDC_STOP)->EnableWindow(TRUE);
}
```

双击IDC_STOP按钮，完成该按钮的消息函数：

```
void CMultiThread1Dlg::OnStop()
{
  // TODO: 在此处添加控制通知句柄代码
  m_bRun=FALSE;
  GetDlgItem(IDC_START)->EnableWindow(TRUE);
  GetDlgItem(IDC_STOP)->EnableWindow(FALSE);
}
```

编译并运行该线程。

### 3. 纤程

纤程（Fiber）是微软为方便将UNIX代码移植到Windows中而添加的一个东西。如果有设计得更好的使用Windows自身线程的应用程序，那么应该避免使用纤程。

从执行体结构上说，纤程和线程是一样的，不同点在于纤程是只能被应用程序手工调度的执行单元。

实现线程的是Windows CE内核。操作系统清楚地知道线程的情况，并且根据系统定义的算法对线程进行调度。纤程是以用户方式代码来实现的，内核并不知道纤程，并且它们是根据

用户定义的算法来调度的。由于用户定义了纤程的调度算法，因此，就内核而言，纤程采用非抢占式调度方式。

单线程可以包含一个或多个纤程。就内核而言，线程是抢占调度的，是正在执行的代码。然而，线程每次执行一个纤程的代码需要决定究竟执行哪个纤程。

## 1.4.3 线程调度

### 1. 线程调度

线程调度采用基于优先级的时间片轮转算法，特点如下所示。

- 时间片大小（Quantum）：在线程获得CPU时间后，会执行特定的一段时间，然后重新调度，这段时间称作时间片大小。
- 默认的时间片是100 ms，OEM可以在OAL中重新设置。
- 获得时间片大小的API函数。

```
DWORD CeGetThreadQuantum(HANDLE hThread);
```

此函数以线程的句柄为参数，返回值是一个32位无符号整数，代表线程的时间片大小。

Windows CE支持抢占式调度，高优先级的线程会抢占低优先级的线程。具有相同优先级的线程平均占有CPU时间片（量程），时间片默认为100ms。OEM可以更改这个值。

线程优先级划分为256个，具体划分如表1-3所示。

- 0最高，255最低。
- 0 ~ 247供驱动和内核使用。
- 应用程序仅能使用248 ~ 255。

表1-3 线程优先级的划分

| 优先级范围 | 分 配 对 象 |
|---|---|
| 0 ~ 96 | 高于驱动程序的程序 |
| 97 ~ 152 | 基于Windows CE的驱动程序 |
| 151 ~ 247 | 低于驱动程序的程序 |
| 248 ~ 255 | 普通的应用程序 |

### 2. 线程调度举例

（1）优先级决定执行顺序

优先级相同的线程轮流运行。如果一个线程正在运行，此时有另一个优先级较高线程要运行，则低优先级线程将立即挂起，转为调度高优先级线程，如图1-8所示。

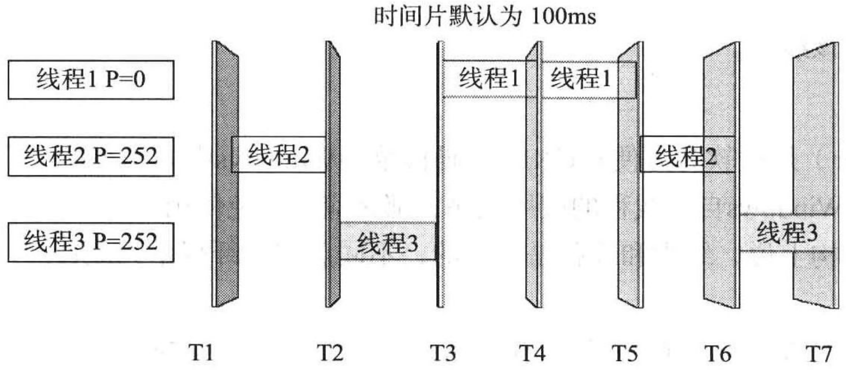



图1-8 优先级决定执行顺序

调度程序中不使用任何老化算法，因此可能出现匮乏（低优先级线程或进程永不运行）。用户需仔细设计应用程序，谨慎分配优先级别，防止出现匮乏现象。

（2）优先级反转

当低优先级线程与高优先级线程争用同一资源时，低优先级线程持有的互斥体或临界区将延迟高优先级线程的执行，此时便会发生优先级反转。为了纠正这种情况并释放高优先级线程，Windows CE通过优先级继承让低优先级线程继承更重要的线程的优先级，然后以较高优先级运行，直至它释放所用资源。优先级继承不适用于信号量，信号量没有与其关联的特定所有者。

由于放弃反转线程所需的时间不确定，且不在内核的控制范围之内，因此OEM将失去对调度进程的控制权。为保证实时性能，OEM应确保不会出现优先级反转条件。

（3）优先级反转示例

假设应用程序有三个线程：线程1具有高优先级；线程2具有中优先级；线程3具有低优先级，如图1-9所示。

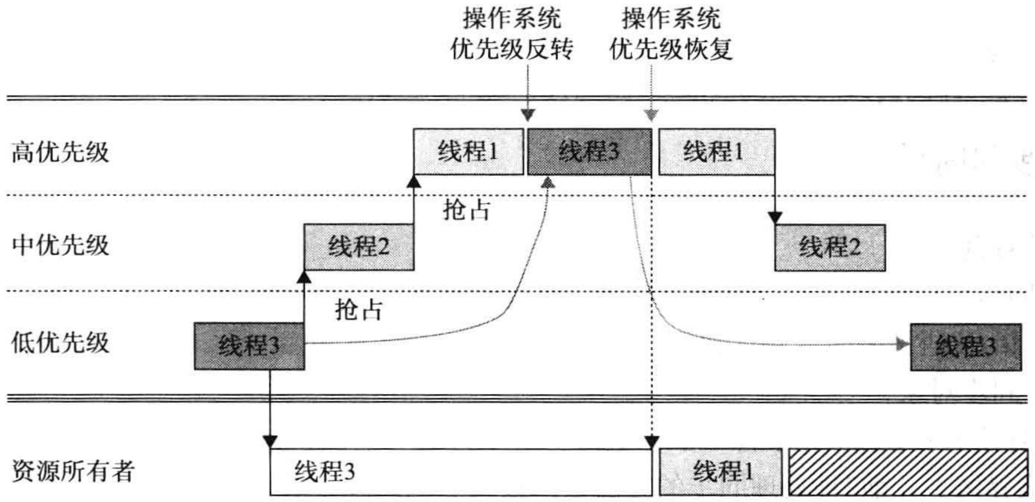



图1-9　优先级反转示例

开始时，线程3运行并进入临界区。此时线程2开始运行，抢占线程3。这样，线程3继续拥有临界区。

接着，线程1开始运行，抢占线程2。线程1尝试进入线程3拥有的那个临界区。但由于该临界区为线程3所有，因此线程1会被阻塞，等待该临界区可用。

此时线程2将开始运行，因为它的优先级比线程3高，且线程1未运行。线程3不会释放线程1正在等待的临界区，因此线程2会继续运行。这样，系统中优先级最高的线程（线程1）处于阻塞状态，等待较低优先级线程的执行。

为解决线程的上述问题，Windows CE支持深度为一个级别的优先级继承。在上例中，当线程1因为等待线程3完成而受阻时，Windows CE会提升线程3的优先级。当线程3的优先级得到提升时，它将运行并最终向线程1释放共享资源。

在线程3释放共享资源之后，Windows CE会将线程3恢复到原始优先级，然后线程1会运行。

但如果线程3受阻，等待另一个线程X释放对象，则Windows CE不会提升线程X的优先级，因为它可能是优先级最低的线程。支持多级别优先级反转对于实时性能有负面影响。如果某个线程的优先级发生反转，则当其优先级不再反转时，将接收到新的量程。

#### 3. 与线程调度相关的API函数

- 读取普通线程优先级

```
int GetThreadPriority(HANDLE hThread);
```

- 设置普通线程优先级

```
BOOL SetThreadPriority(HANDLE hThread,int  nPriority);
```

- 获取实时线程优先级

```
int CeGetThreadPriority(HANDLE hThread);
```

- 设置实时线程优先级

```
BOOL CeSetThreadPriority(HANDLE hThread,int nPriority);
```

- 查询线程的时间片

```
DWORD CeGetThreadQuantum(HANDLE hThread);
```

- 设置线程的时间片

```
BOOL CeSetThreadQuantum(HANDLE hThread,DWORD dwTime);
```

### 1.4.4 线程同步

#### 1. 线程同步

线程同步可以保证在一个时间内只有一个线程对某个资源有控制权。同步还可以使得有关联交互作用的代码按一定的顺序执行。

（1）同步对象

同步对象有：Event（事件）、Mutex（互斥对象）、Semaphores（信号量）、Critical_section（临界区）。前3种处理同步的方法是内核态下的同步方法。

这些对象有两种状态：通知（signaled）或未通知（not signaled）状态。

线程通过同步等待函数来使用同步对象。一个同步对象在同步等待函数调用时被指定，调用同步函数的线程被阻塞（blocked），直到同步对象获得通知。被阻塞的线程不占用CPU时间。

（2）等待函数API

等待函数有四个，接下来分别介绍。

- 等待单个对象，代码如下：

```
DWORD WaitForSingleObject(同步对象句柄,等待时间)
DWORD WaitForSingleObject(
  HANDLE  hHandle,                //同步对象的句柄
  DWORD dwMilliseconds            //等待的ms数
);
```

- 等待多个对象，代码如下：

```
DWORD WaitForMultipleObjects(同步数,同步对象句柄数组,FALSE,等待时间)
DWORD WaitForMultipleObjects(
```

```
  DWORD    nCount,                  //lpHandles中的同步对象个数
  CONST HANDLE* lpHandles,          //同步对象句柄的数组
  BOOL  fWaitALL,                   //是否等等该lpHandles中所有的内核对象
  DWORD dwMilliseconds              //等待的ms数
);
```

● 如果一个线程既要执行大量任务又要响应用户的按键消息，可以采用两个专用于等待消息的函数，代码如下：

```
DWORD MsgWaitForMultipleObjects(DWORD nCount,
LPHANDLE pHandles,BOOL fWaitAll,DWORD dwMilliseconds,
DWORD dwWakeMask);
DWORD MsgWaitForMultipleObjectsEx(DWORD nCount,
LPHANDLE pHandles,DWORD dwMilliseconds,DWORD
dwWakeMask,DWORD dwFlags);
```

### 2. 事件

当一个线程需要通知其他线程发生了什么时，可用事件同步对象。

（1）处理过程

首先，调用CreateEvent函数创建一个事件对象，该函数返回一个事件句柄。然后，可以设置（SetEvent）或者复位（ResetEvent）一个事件对象，最后，使用CloseHandle销毁创建的事件对象。

复位有两种形式：自动复位和人工复位。在创建事件对象时指定复位形式。

● 自动复位：当对象获得通知后，就释放下一个可用线程（优先级最高的线程；如果优先级相同，则等待队列中的第一个线程被释放），系统随后会自动将状态重置为无信号。
● 人工复位：当对象获得通知后，就释放所有可利用线程，随后会自动将状态重置为无信号。

（2）函数

创建Event同步对象，代码如下：

```
HANDLE WINAPI CreateEvent(
LPSECURITY_ATTRIBUTES lpEventAttributes,        //不支持，设为NULL
BOOL bManualReset,                              //是否需要人工重置
BOOL bInitialState,                             //初始状态，TRUE为通知状态
LPCTSTR lpName;                                 //可选，Event的名字
);
```

把Event同步对象设置为通知状态，代码如下：

```
BOOL  SetEvent(
HANDLE hEvent                                   //Event的句柄
);
```

把Event同步对象设置为未通知状态，代码如下：

```
BOOL ResetEvent(
HANDLE hEvent                                   //Event的句柄
);
```

（3）示例

使用Event对象对共享的全局变量i进行访问，代码如下：

```
HANDLE     hMutex;
```

```
INT             i=0;
//创建一个自动重置,初始状态为通知的Event对象。创建线程
VOID Init()
{     hEvent = CreateEvent(NULL,FALSE,TRUE,NULL);
      CreateThread(NULL,NULL,ThreadProc1,NULL,NULL,NULL);
      CreateThread(NULL,NULL,ThreadProc2,NULL,NULL,NULL);}
DWORD   WINAPI   ThreadProc1(LPVOID lpParameter)
{      WaitForSingleObject(hEvent,INFINITE);     //等待事件发生
      i++;                                       //访问共享资源
      SetEvent(hEvent);                          //让事件再次发生
      return 0;}
DWORD   WINAPI   ThreadProc2(LPVOID lpParameter)
{     WaitForSingleObject(hEvent,INFINITE);      //等待事件发生
      i--;                                       //访问共享资源
      SetEvent(hEvent);                          //让事件再次发生
      return 0;}
```

### 3. 互斥对象（Mutex）

在一个线程访问某个共享资源时，互斥对象（Mutex）能够保证其他线程不能访问这个资源。

（1）处理过程

首先调用CreateMutex（NULL，是否属于该线程，互斥对象名）创建互斥对象，然后调用等待函数，最后调用ReleaseMutex释放互斥对象。

（2）函数

创建一个Mutex同步对象，代码如下：

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,    //不支持,设置为NULL
    BOOL bInitialOwner,                         //是否为创建该Mutex的线程所拥有
    LPCTSTR lpName                              //可选,该Mutex的名字
);
```

释放对Mutex的占用，代码如下：

```
BOOL ReleaseMutex(
    HANDLE hMutex                               //Mutex的句柄
);
```

例如，两个线程对全局变量i的同步访问，代码如下：

```
HANDLE hMutex;
INT i=0;
VOID Init() //创建互斥对象和线程
{   hMutex = CreateMutex(NULL,FALSE,NULL);
    CreateThread(NULL,NULL,TPRO1,NULL,NULL,NULL);
    CreateThread(NULL,NULL,TPRO2,NULL,NULL,NULL);
}
DWORD WINAPI TPRO1(LPVOID  lpPara)
{ WaitForSingleObject(hMutex,INFINITE);
i++
  ReleaseMutex(hMutex);
  return 0;
 }
DWORD WINAPI TPRO2(LPVOID  lpPara)
```

```
{ WaitForSingleObject(hMutex,INFINITE);
  i--
  ReleaseMutex(hMutex);
  return 0;
}
```

### 4. 临界区

临界区对象能保证在临界区内所有被访问的资源不被其他线程访问，直到当前线程执行完临界区代码。

与互斥对象的主要区别：互斥对象有句柄，可以在进程间使用，是内核态下的同步方法；但临界区对象只能用于同一进程的线程之间，是用户态下的同步方法。

（1）处理过程

首先定义一个临界区对象cs：`CRITICAL_SECTION cs;`

然后初始化该对象：`InitializeCriticalSection(&cs);`

在进入临界区代码前调用EnterCriticalSection函数，这样其他线程都不能执行该段代码，若它们试图执行就会被阻塞。

完成临界区的执行之后，调用LeaveCriticalSection函数，其他的线程可以继续执行该段代码。如果不调用该函数，则其他线程将无限期等待。

最后删除临界区：`DeleteCriticalSection`

（2）函数

- InitializeCriticalSection：分配CRITICAL_SECTION结构。
- EnterCriticalSection：调用，在占有cs的线程调用LeaveCriticalSection之前会阻塞。
- TryEnterCriticalSection：EnterCriticalSection的非阻塞版。
- LeaveCriticalSection：释放CriticalSection的所有权。
- DeleteCriticalSection：释放InitializeCriticalSection分配的资源。

例如：

```
CRITICAL_SECTION cs1,cs2;                    //定义临界区对象
INT i=0;
VOID Init()
{   InitializeCriticalSection(&cs1);        //初始化临界区变量
    InitializeCriticalSection(&cs2);
    CreateThread(NULL,NULL,TPRO1,NULL,NULL,NULL);
    CreateThread(NULL,NULL,TPRO2,NULL,NULL,NULL);
}
DWORD WINAPI TPRO1(LPVOID  lpPara)
{ EnterCriticalSection(&cs1);               //开始保护机制
  i++
  LeaveCriticalSection(&cs1);               //撤销保护机制
  return 0;
}
DWORD WINAPI TPRO2(LPVOID  lpPara)
{ EnterCriticalSection(&cs2);
  i--
```

```
    LeaveCriticalSection(&cs2);
    return 0;
}
```

### 5. 信号量

信号量（Semaphore）用于限制资源访问数量，其中包含一个引用计数、一个当前可用资源数和一个最大可用资源数。如果当前可用资源数大于0，信号量对象处于通知状态；当可用资源数等于0，信号量对象处于未通知状态。

（1）处理过程

首先调用CreateSemaphore（NULL，初始计数，最大线程数，信号量名）创建一个信号量；然后调用等待函数，如果允许的话，则利用关键资源；最后调用ReleaseSemaphore（信号量对象句柄，释放的引用数，返回释放前的信号量计数）释放信号量对象。

（2）函数

创建一个Semaphore对象，代码如下：

```
HANDLE WINAPI CreateSemaphore(
 LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,     //不支持,设为NULL
 LONG lInitialCount,                              //初始计数
 LONG lMaximumCount,                              //最大支持的线程数
LPCTSTR lpName                                    //可选,Semaphore的名字
);
```

释放Semaphore，代码如下：

```
BOOL ReleaseSemaphore(
 HANDLE hSemaphore,        //要释放的Semaphore句柄
 LONG lReleaseCount,       //释放的引用个数
 LPLONG lpPreviousCount    //返回释放前的Semaphore计数
);
```

## 1.4.5 进程间通信

进程间通信是指不同进程之间进行数据共享和数据交换。

Windows CE提供了多种进程间通信的方式，如粘贴板、共享堆、文件映射、DDE（动态数据交换）、匿名管道、命名管道、邮件箱、对象连接与嵌入、远程过程调用、Sockets、WM_COPYDATA消息等。

### 1. 文件映射

文件映射（Memory-Mapped File）能使进程把文件内容当做进程地址区间一块内存那样来对待。因此，进程不必使用文件I/O操作，只需简单的指针操作就可读取和修改文件的内容。

Win32 API允许多个进程访问同一文件映射对象，各个进程在其地址空间里接收内存的指针。通过使用这些指针，不同进程就可以读取或修改文件的内容，实现对文件中数据的共享。代码如下所示：

```
HANDLE hFile,hMap;   LPVOID hMMF;
hFile = CreateFileForMapping(...);    //创建文件,用于内存映射
//创建内存映射文件
```

```
//当hFile等于INVALID_HANDLE_VALUE时,可获得内存映射文件
hMap = CreateFileMapping(hFile, ...);
if(hMap != NULL && GetLastError() == ERROR_ALREADY_EXISTS)
{CloseHandle(hMap);   hMap = NULL; }
//把内存映射文件映射到进程共享地址空间
//参数: 内存映射文件对象,权限,起始映射地址偏移的高32位,起始映射地址偏移的低32位,要映射的字节数(0表示全部)
hMMF = MapViewOfFile(hMap,FILE_MAP_ALL_ACCESS,0,0,0)
//读取内存映射文件的内容
int intV= *((int *)hMMF);
//写数据去内存映射文件
*((int *)hMMF)=12;
//清理资源
UnmapViewOfFile(hMMF);
CloseHandle(hMap);
CloseHandle(hFile);
```

文件映射是在多个进程间共享数据的非常有效的方法，有较好的安全性。但文件映射只能用于本地机器的进程之间，不能用于网络中，而开发者还必须控制进程间的同步。

#### 2. 点对点消息队列

Windows CE允许一个应用程序或驱动程序创建自己的消息队列。消息队列既可以作为在线程之间传递数据的工具，也可以作为线程之间同步的工具。它的优点是只需很小的内存，一般只用于点到点的通信。

消息队列的特点如下所示：

- 消息可为任意数据类型。
- 消息队列可用来进行同步。
- 消息无优先级，满足FIFO特性。
- Windows CE消息队列是基于点对点的，不能用来广播通信。

与消息队列相关的API函数：

- CreateMsgQueue：创建或打开用户定义的消息队列。
- OpenMsgQueue：打开现有消息队列的句柄。
- CloseMsgQueue：关闭打开的消息队列。
- ReadMsgQueue：从消息队列读取一条消息。
- WriteMsgQueue：向消息队列写入一条消息。
- GetMsgQueueInfo：返回有关消息队列的信息。

## 1.5 中断处理

### 1.5.1 中断基本概念

大多数的外部设备会产生中断来请求操作系统的服务。一般情况下，外部设备并不占用CPU，只有当外设工作时，才会通过中断来打断CPU的当前执行，请求CPU处理外部设备的数据。

因为外设会产生中断，所以外设相对应的驱动程序就要处理这些中断。

物理中断请求（Interrupt ReQuest，IRQ）是外部设备通过CPU的中断引脚向CPU发送的中断信号。

逻辑中断（SYSINTR）是对物理中断的抽象（ISR），例如，不同开发板上的键盘产生的IRQ中断，逻辑中断标识为SYSINTR_KEYBOARD。

中断只是异常处理的一部分，特点如下：

- 在内核代码中被获取。
- 在恢复到用户态之前进行处理。
- 硬件中断是独立于操作系统的。

Windows CE使用两阶段中断服务模式：

- 少量的代码进行中断的记录。
- 调度中断服务程序代码进行处理。

操作系统提供API来使能IRQ信号，设置中断的优先级。

与硬件有关的操作在OAL以及设备驱动代码中进行。

当一个中断发生之后，操作系统陷入异常，内核将调用该中断的注册ISR。ISR首先屏蔽该中断，处理后返回相应的中断号（SYSINTR）。系统得到该中断号之后，就会找到该中断号对应的事件，并唤醒等待相应事件的线程（IST），然后IST进行中断处理，处理完成之后，IST通知内核重新启用硬件中断。

Windows Embedded CE 6.0的中断处理过程如图1-10所示。

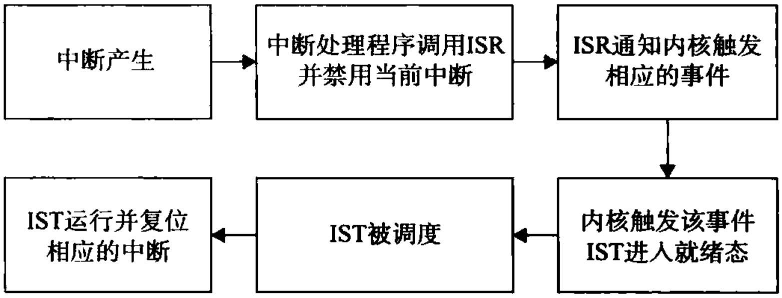



图1-10 中断处理的过程

Windows CE的一次中断处理起始于硬件中断，终止于IST调用InterruptDone函数。

在其他的一些嵌入式操作系统中，在介绍中断处理时经常会提到一个中断向量表的概念，如uC/OS。当中断发生时它会调入IRQ的处理程序，并根据IRQ的值跳转到事先分配好的中断向量表相应的中断处理函数中。但在Windows CE中实际上并不存在中断向量表的概念，而只有一个异常向量表，对应于MCU的几种运行模式。Windows CE的中断处理对应于两个异常IRQHandler和FIQHandler，通常使用的是IRQHandler。当产生外部中断时，系统执行IRQHandler函数，在IRQHandler函数中调用OEMInterruptHandler函数获取IRQ对应的SYSINTR，然后根据SYSINTR调用函数OEMNotifyIntrOccurs函数触发与SYSINTR关联的事

件，最后由IST完成主要的中断处理工作。这种中断处理机制在一定程度上影响了系统的实时性，但提高了IST的灵活性。

## 1.5.2 中断服务例程

中断服务例程（ISR）是内核中的低级处理程序，中断发生时首先被调用。

ISR的实现在OAL（OEM适配层）中，如图1-11所示。它只处理最低级的中断响应，通常用来获取IRQ和SYSINTR并设置MCU内部的中断控制寄存器。中断处理的主要部分在驱动或者应用的中断处理线程中。中断处理线程与其他普通线程一样，使用同一个线程优先级管理系统。ISR和IST之间通过事件对象进行同步。

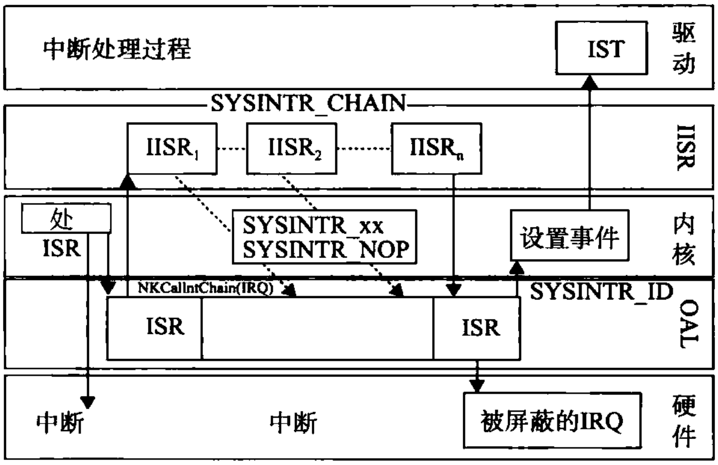



图1-11 ISR结构图

### 1. ISR可能的返回值

- SYSINTR_NOP：中断不与设备的任何已注册ISR关联。内核启用所有其他中断。
- SYSINTR_XXX（逻辑中断号）：中断与已知的已注册ISR和设备关联。
- SYSINTR_RESCHED：中断是由请求操作系统重新调度的计时器到期引起的。
- SYSINTR_RTC_ALARM：闹钟已到期。

### 2. ISR检查中断并决定如何处理中断

- 如果中断需要被内核和IST进一步处理，那么ISR会返回一个逻辑中断号SYSINTR_XXX。
- 如果该中断不需要进一步的处理，那么ISR只需返回SYSINTR_NOP给内核。

从Windows CE 4.0开始，微软引入了IISR（Installabel Interrupt Service Routine），以便应用程序开发商可以安装自己的ISR处理外设的中断。有了IISR，多个设备可以共享同一个硬件IRQ。

### 3. 安装ISR到平台的两个步骤

- 调用LoadIntChainHandler函数以加载包含ISR代码的DLL。
- 该ISR须把IRQ转化为某个SYSINTR_XXX并返回。

#### 4. 创建可安装的ISR的基本外壳程序

- DLLEntry：接收进程和线程附加消息 。
- InfoCopy：在进行任何结构赋值时使用的复制例程 。
- IOControl：任何使用KernelLibIOControl的IST调用的处理程序 。
- ISRHandler：实际的ISR。

### 1.5.3 中断服务线程

中断服务线程（IST）是处于驱动或者应用程序中的中断处理线程，由系统调度，完成大部分的中断处理工作。在IST中创建一个事件对象，并使用WaitForSingleObject函数等待该事件被触发，在ISR中通知内核触发相应的事件对象。在使用IST时应注意以下几点：

- IST线程大多数时候是空闲的，只有操作系统通知IST有中断发生时，IST才开始工作。
- 通过把一个逻辑中断号与一个Win32的同步对象事件相关联来实现，当中断发生时，操作系统会引发与该逻辑中断相关联的事件。
- IST是内置设备的驱动程序的用户态线程，用来进行实际的中断处理。
- 调用CreateEvent创建一个与逻辑中断相关联的事件对象。
- IST在大多数时间里都处于空闲状态，在内核用信号通知事件对象时被唤醒，IST通常运行的优先级高于普通优先级，其优先级由CeSetThreadPriority函数设定。

#### 1. 初始化中断的关键步骤

- 创建事件。
- 获取IRQ的逻辑中断号。
- 创建挂起的中断线程（IST）。
- 调用InterruptInitialize以创建IRQ与事件之间的关联，创建未挂起的IST可能会导致InterruptInitialize失败。
- 该事件已经处于被等待状态。
- 设置线程的优先级。
- 恢复IST。

#### 2. InterruptInitialize函数

此函数负责把某个逻辑中断号与一个Event内核对象关联起来。代码如下：

```
BOOL InterruptInitialize(
         DWORD idInt,                    //SYSINTR中断号
         HANDLE hEvent,                  //与该中断相关联的事件句柄
         LPVOID pvData,                  //传给OEMInterruptEnable函数的缓冲区指针
         DWORD cbData                    //缓冲区的大小
);
```

#### 3. WaitForSingleObject函数

此函数阻塞当前的线程，等待某个Event内核对象标识的事件发生。这一调用通常放在一

个循环中，在中断处理时，使IST可以回到这一个调用，等待下一个要处理的中断。函数声明如下所示：

```
DWORD WINAPI WaitForSingleObject(
HANDLE hHandle,              //事件的句柄
//超时的ms数,如果超过此参数设定的时间,那么函数也会返回
DWORD dwMilliseconds);
```

### 4. InterruptDone函数

此函数用来告诉操作系统对该中断的处理已完成，操作系统可重新开启该中断。中断处理完以后，IST必须调用InterruptDone函数来重新开启该中断。

```
VOID InterruptDone
(
DWORD idInt            //SYSINTR逻辑中断号
);
```

处理来自应用程序或驱动程序的中断需要进行两个步骤。首先，必须使用关联的事件初始化中断；其次，IST必须等待中断事件以响应内核中的中断。

### 5. 示例

创建一个线程，该线程的任务就是等待中断，设置等待时间为50ms，如果50ms以后线程未返回，就调用TerminateThread强制终止此线程，如果线程正常返回就将数值写入缓冲区，等待上层应用程序的调用。以下是部分代码：

```
HANDLE gADCIntrThread = CreateThread(0,
0,(LPTHREAD_START_ROUTINE)
ADC_IntrThread,0,0,&IDThread); /* 创建线程,用于等待中断*/
gADCIntrEvent =CreateEvent(NULL,FALSE,TRUE,NULL);
InterruptInitialize(SYSINTR_ADC,gADCIntrEvent,0,0); /*将事件与中断关联*/
TerminateThread(g_htIST,);/* 强制终止此线程*/
WaitForSingleObject(gADCIntrThread,50);/* 等待线程返回*/
```

# 第2章　搭建Windows CE开发环境

从6.0版开始，微软将Windows CE的名字改为Windows Embedded CE，同时也将Platform Builder开发环境整合成为Visual Studio的一个插件。因此为了建立Windows Embedded CE 6.0的开发环境，需要准备好由微软公司发布的Visual Studio 2005（英文版的主程序共有两张光盘，而MSDN则有三张光盘）及Windows Embedded CE 6.0安装光盘（一张DVD光盘）。这几张光盘包含了微软公司的整合开发环境Visual Studio 2005、Windows Embedded CE 6.0操作系统安装程序，以及将操作系统编译移植到指定目标硬件平台的平台建立器（Platform Builder for CE 6.0）。下面介绍如何从光盘安装整合开发环境Visual Studio 2005和Windows Embedded CE 6.0操作系统。

## 2.1　安装Visual Studio 2005

### 2.1.1　安装Visual Studio 2005

首先必须先安装整合开发环境Visual Studio 2005。将标识为Disk 1的光盘放入PC光驱中，将会自动开启安装程序，如图2-1所示。如果在Windows中将光盘的自动执行（Autorun）功能关闭，请开启文件管理进入光盘目录内执行Autorun.exe这个文件。

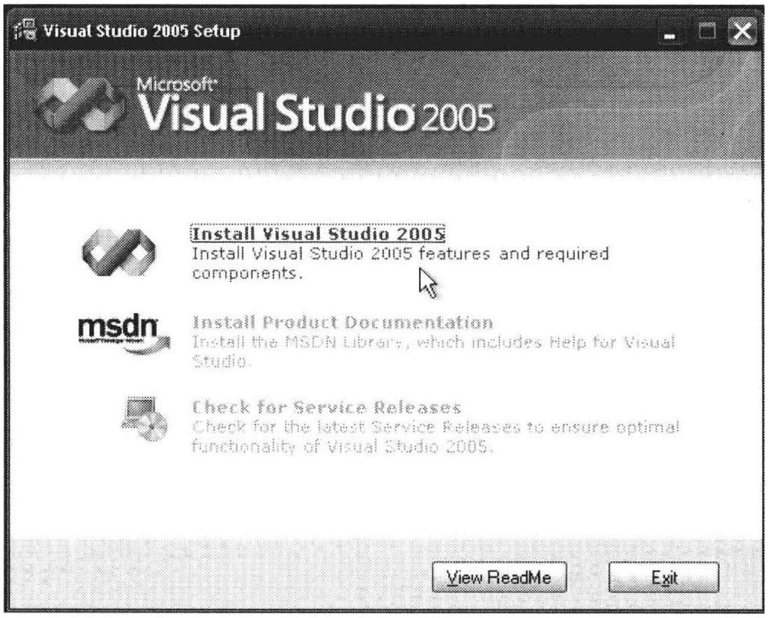



图2-1　Visual Studio 2005安装界面

单击“Install Visual Studio 2005”开始安装。建议在安装Visual Studio 2005前确认系统中已安装Windows Installer 3.1。若系统尚未安装Windows Installer 3.1，在接下来的安装过程中，程序将会自动先安装Windows Installer 3.1，再安装Visual Studio 2005。

在安装过程中需要填入产品序列号（Product key），可以在微软的产品包装上找到产品序列号，将它填入后单击“Next”按钮继续安装，如图2-2所示。

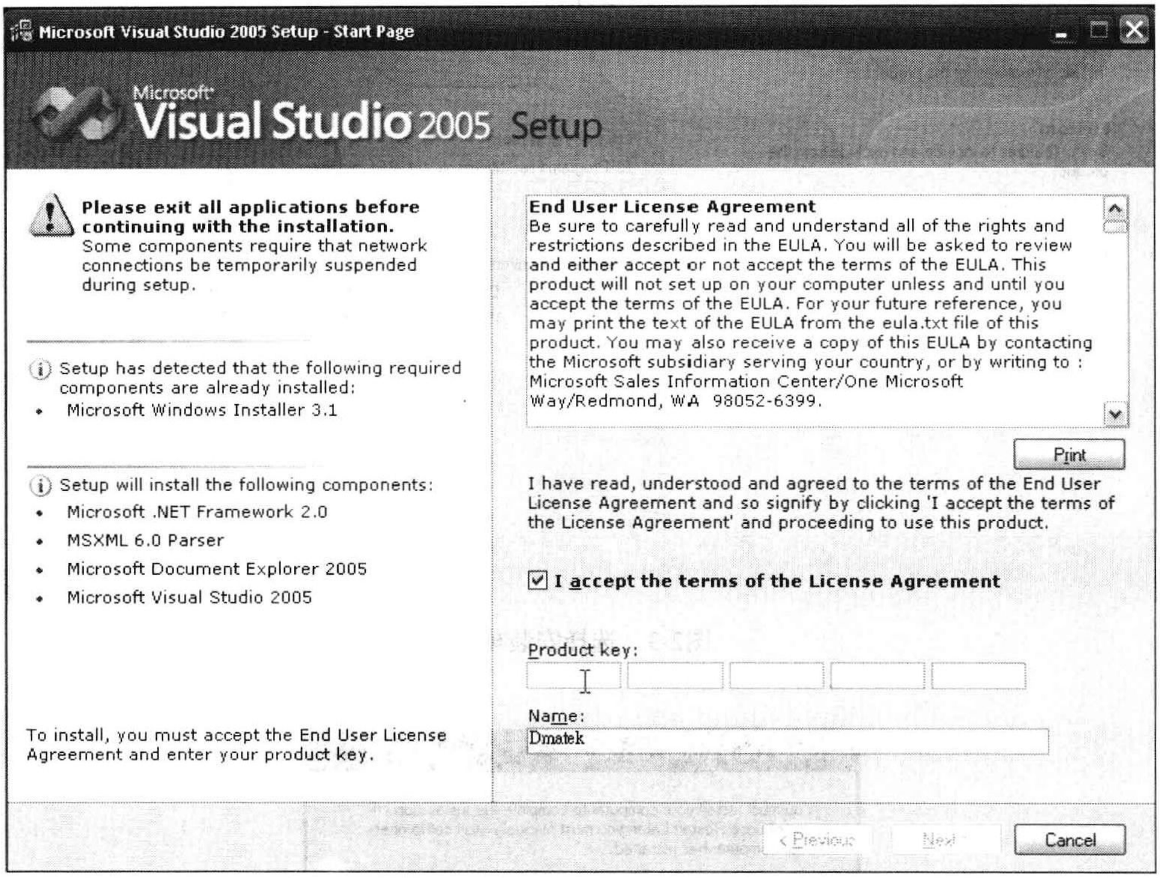



图2-2 安装过程中需要填入产品序列号

在安装过程中，可以选择要安装的程序开发语言及组件，如果不打算安装MSDN，则至少需要2GB的硬盘空间，并且必须保留至少1GB的剩余空间。如果决定安装MSDN，完整安装MSDN则需要3.8GB的硬盘空间，并至少保留1GB的剩余空间。

一般采用默认（Default）安装就可以了，若硬盘空间足够，建议采用完整（Full）安装模式，此时大约需要3.2GB的硬盘空间，如图2-3所示。

在安装过程中，视系统的状况可能会要求重新启动计算机，如图2-4所示，单击“Restart Now”按钮重新启动计算机。此时光驱中的光盘不必退出，在系统重新启动完毕后，安装程序将自动继续安装其他未装好的组件。

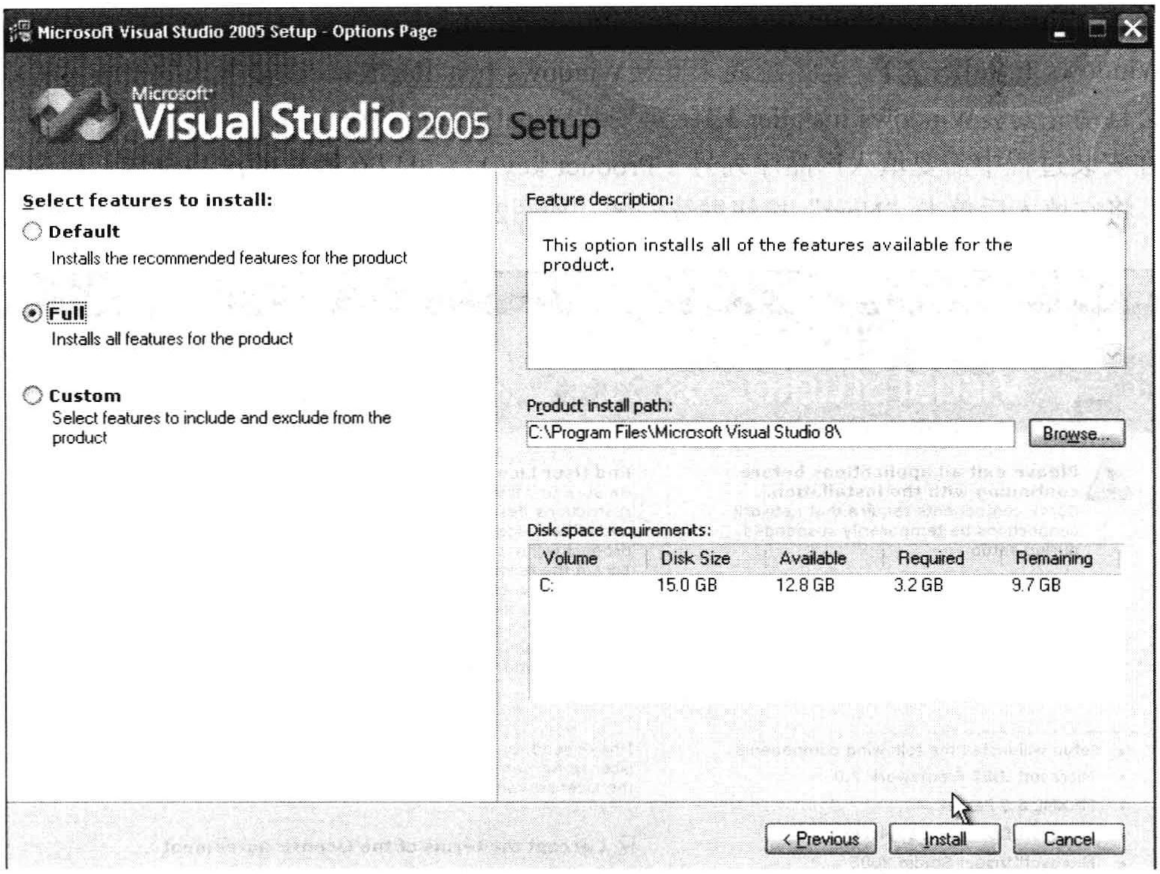



图2-3 选择安装模式

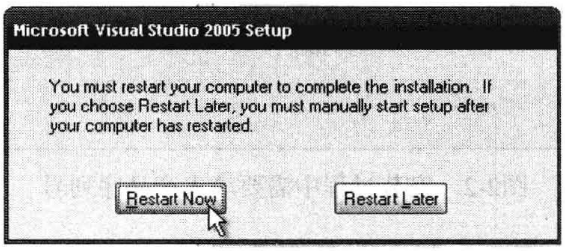



图2-4 安装结束后重启系统

在安装过程中会提示更换第二张光盘。当Visual Studio 2005全部安装完毕后，会出现如图2-5所示的窗口，建议到微软公司的网站下载最新的修正程序。单击“Finish”按钮结束安装过程。

待Visual Studio 2005主程序安装完毕后，可以视需要决定是否安装Product Documentation（MSDN）。安装完毕后，在“开始”菜单的“程序”中找到“Microsoft Visual Studio 2005”并执行Microsoft Visual Studio 2005，第一次启动Visual Studio 2005时会要求选择一个默认的环境设定（Default Environment Settings），由于此时尚未安装Windows Embedded CE 6.0，因此这里选择“Visual C++ Development Settings”，如图2-6所示。

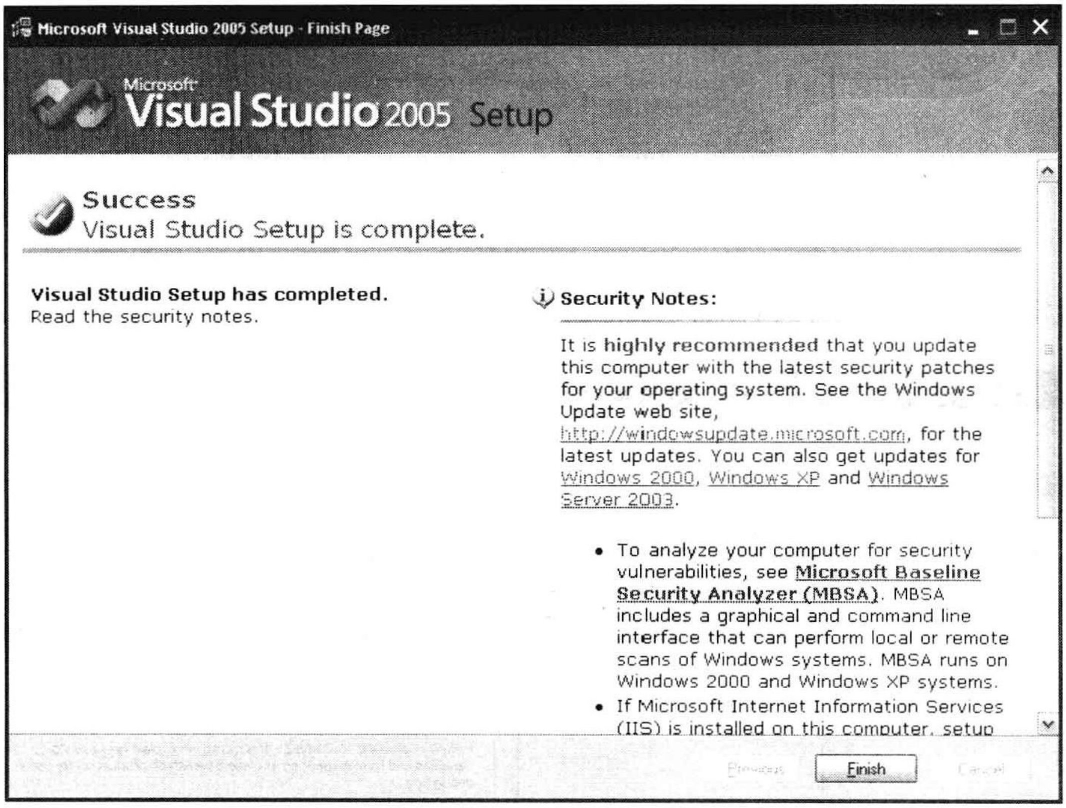



图2-5　结束安装

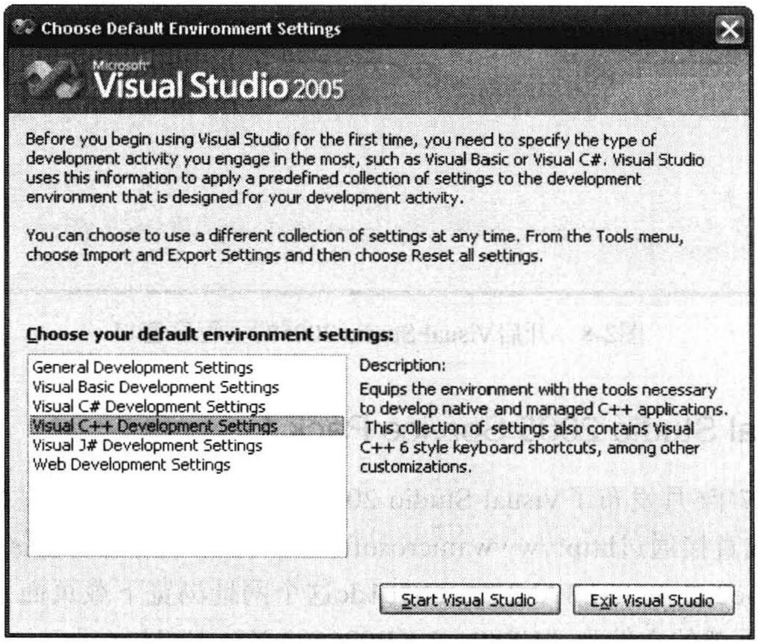



图2-6　默认的环境设定

依机器状况而定，Visual Studio 2005会花数秒到数分钟的时间建立环境设定，如图2-7所示。

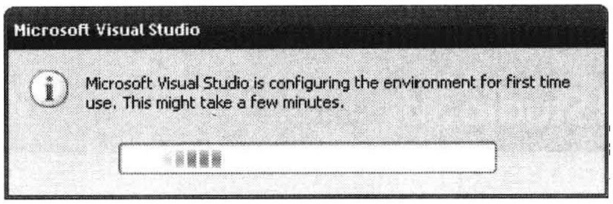



图2-7 默认环境的配置

当设定完成后，就会开启Visual Studio 2005的主程序窗口，如图2-8所示。

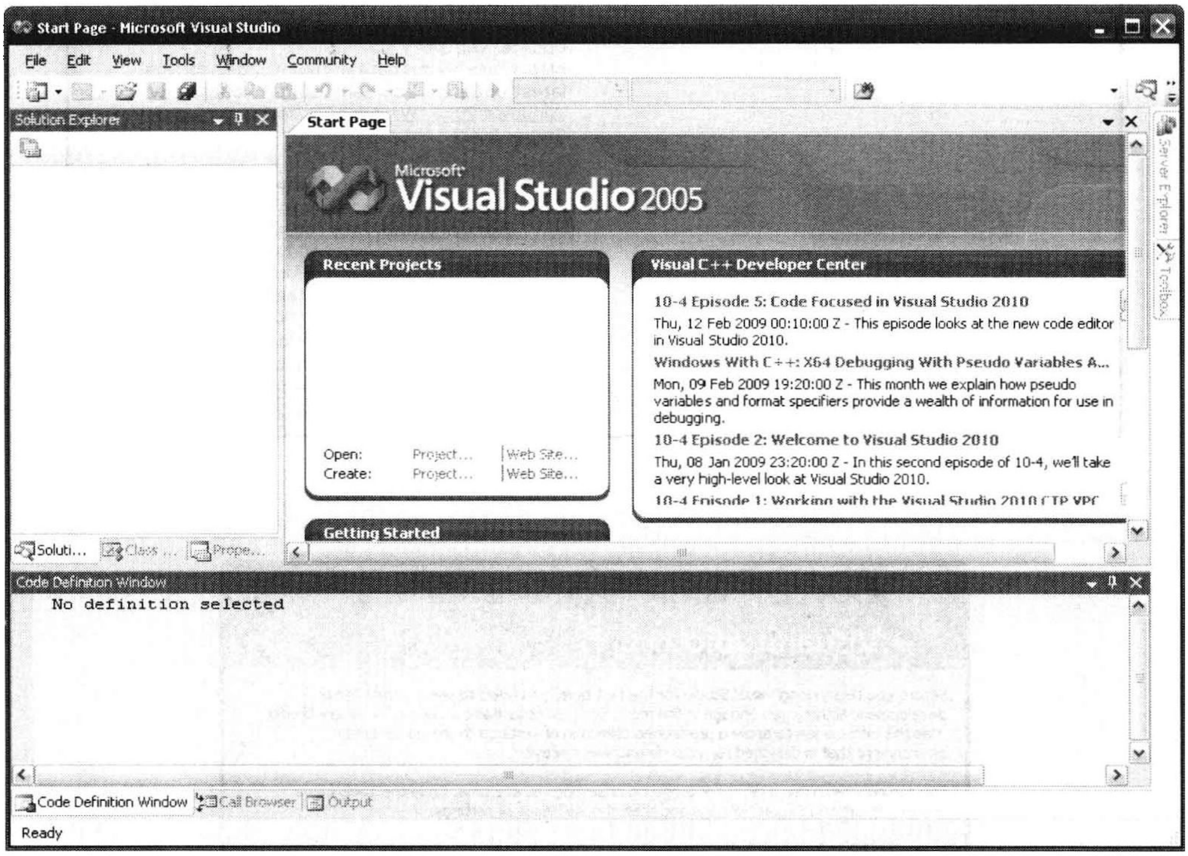



图2-8 开启Visual Studio 2005的主程序窗口

## 2.1.2 安装Visual Studio 2005 Service Pack 1

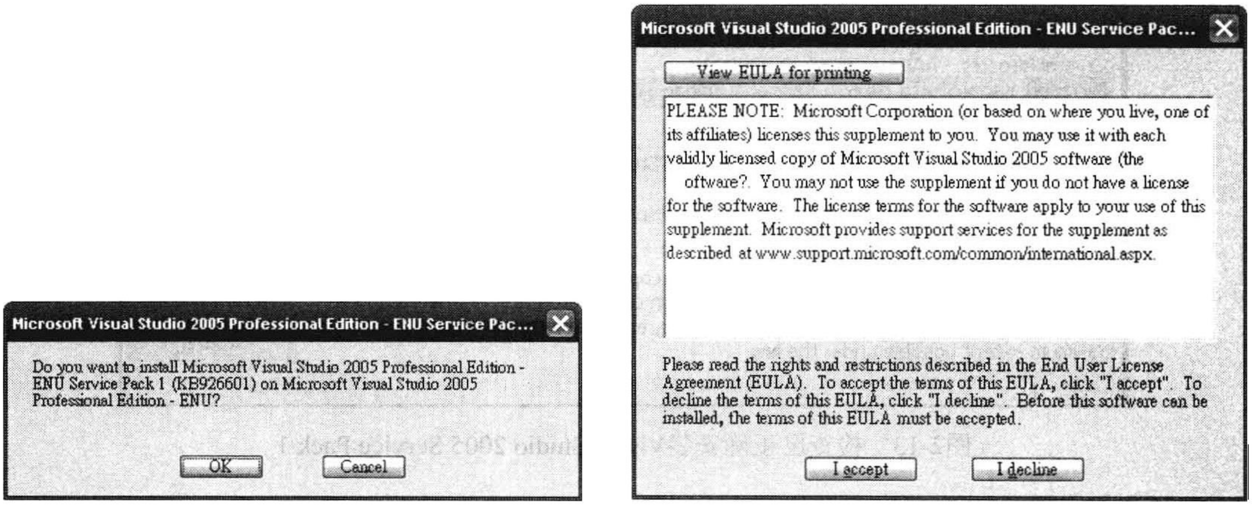
微软公司在2007年6月发布了Visual Studio 2005 Service Pack 1，可以在微软下载中心的网站找到下载页面，或直接通过http://www.microsoft.com/ downloads/details.aspx?displaylang=en&FamilyID=bb4a75ab-e2d2-4c96-b39d-37baf6b5b1dc这个网址浏览下载页面，本书采用英文版的Visual Studio，因此下载的文件为“VS80sp2-KB926602-X86-ENU.exe”。

执行VS80sp2-KB926602-X86-ENU.exe，开始安装Visual Studio 2005 Service Pack 1，安装程序会先将需要的文件解压缩并准备进行安装，如图2-9所示。

a）　　　　　　　　　　b）

图2-9　提取数据并准备安装

当准备就绪后，安装程序会提示确认是否安装Service Pack 1，如图2-10所示，单击“OK”按钮继续安装。此外，安装程序也会提示请先阅读EULA，同样请单击“I accept”按钮继续安装。




a）　　　　　　　　　　b）

图2-10　开始安装

接下来开始安装Visual Studio 2005 Service Pack 1，整个安装过程需要较长的时间，请耐心等候安装完毕，如图2-11所示。

当Visual Studio 2005 Service Pack 1安装完毕后，安装程序会提示如图2-12所示的对话框，请单击“OK”按钮结束安装程序。

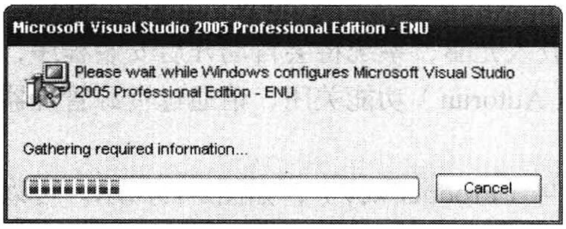



图2-11　安装过程

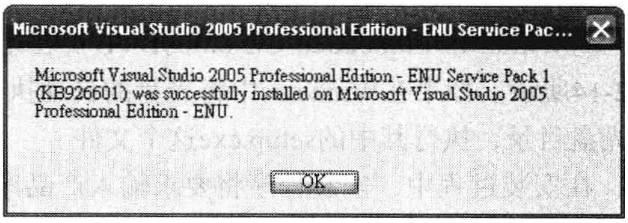



图2-12　结束安装程序

此时若重新开启Visual Studio 2005，则可以在“About Microsoft Visual Studio”对话框中检

查是否已正确安装Visual Studio 2005 Service Pack 1。如图2-13所示。

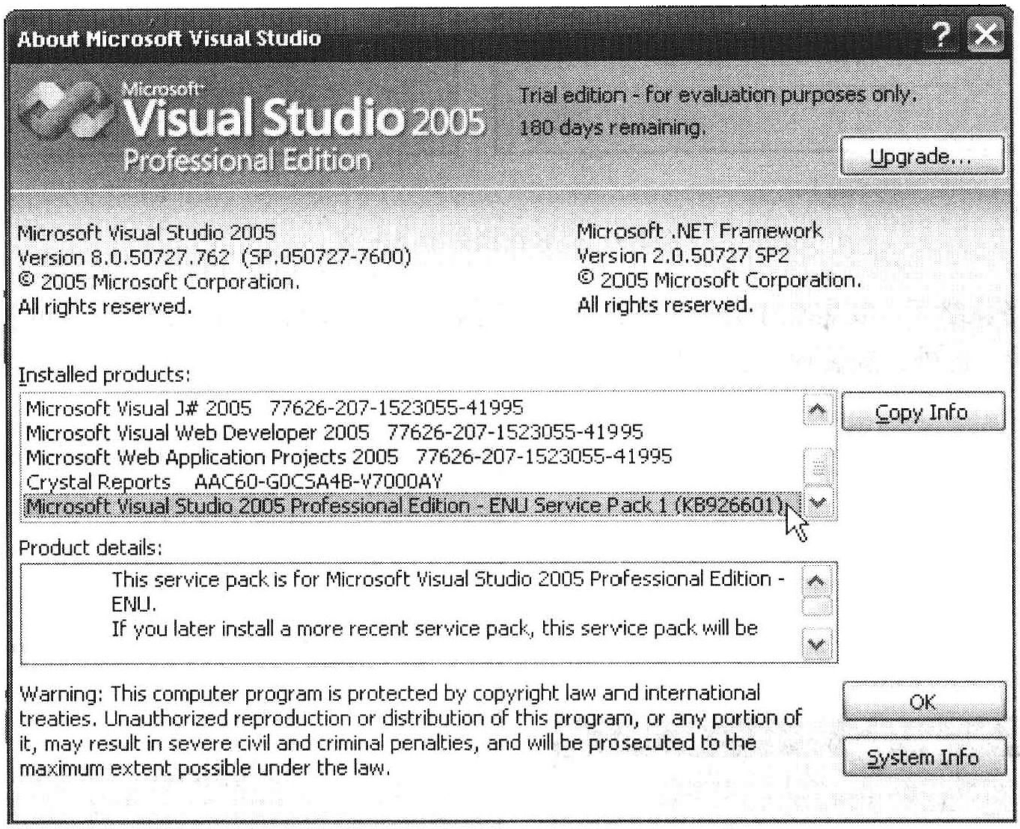



图2-13 检查已正确安装Visual Studio 2005 Service Pack 1

此外，若用户使用的是Windows Vista，则必须再安装Visual Studio 2005 Service Pack 1 Update for Windows Vista，可以从http://www.microsoft.com /downloads/details.aspx?displaylang=en &FamilyID=90e2942d-3ad2-4872-a2ee-4acc0aace5b6这个网页下载安装程序，英文版本的文件名称为VS80sp2-KB932232-X86- ENU.exe。

## 2.2 安装Windows Embedded CE 6.0

安装好Visual Studio 2005及Service Pack 1后，就可以开始安装Windows Embedded CE 6.0了。将Windows Embedded CE 6.0的DVD安装光盘放入光驱，系统将会自动开启安装程序，如图2-14所示。如果在Windows中将光盘的自动执行（Autorun）功能关闭，请通过资源管理器进入光盘目录，执行其中的setup.exe这个文件。

在安装过程中，安装程序将要求输入产品序列号（Product key），如图2-15所示。可以在微软的产品包装上找到产品序列号，将它填入并单击“Next”按钮继续安装程序。

图2-14 开始安装Windows Embedded CE 6.0

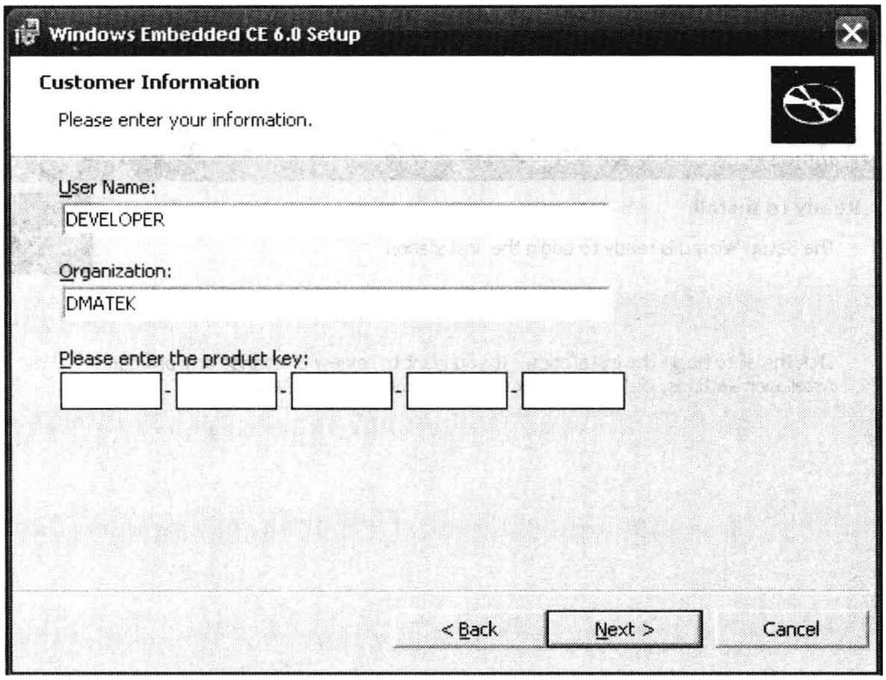



图2-15 输入产品序列号

在安装过程中，可以选择所需的平台，如图2-16所示。请确认必须选取“ARMV4I”这一项。ARMV4I提供了对ARM72X、ARM92X、ARM102X及Intel XScale等内核的支持。图2-16所选择的安装项目在安装完Windows Embedded CE 6.0后，约占用4.2GB的硬盘空间。

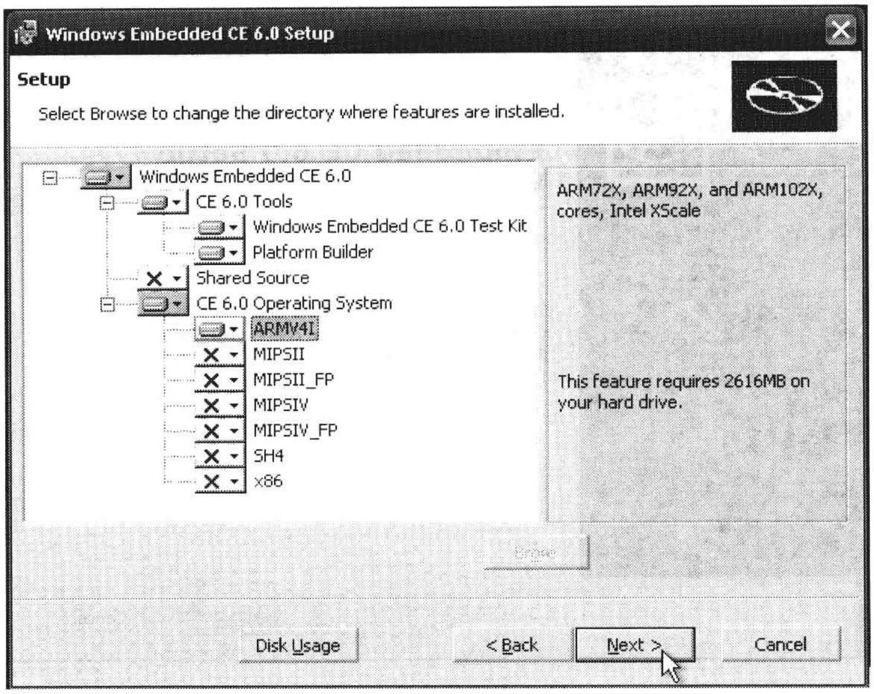



图2-16 选择所需的平台

选择要安装的项目后，单击“Next”按钮开始安装所需要的文件，如图2-17、图2-18所示。

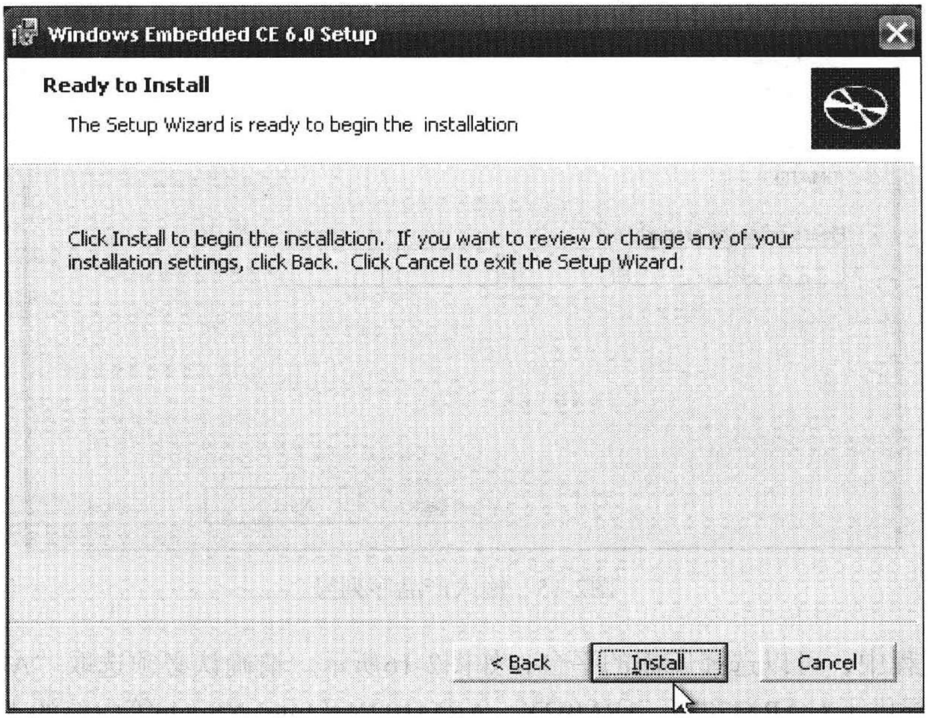



图2-17 开始安装所需要的文件

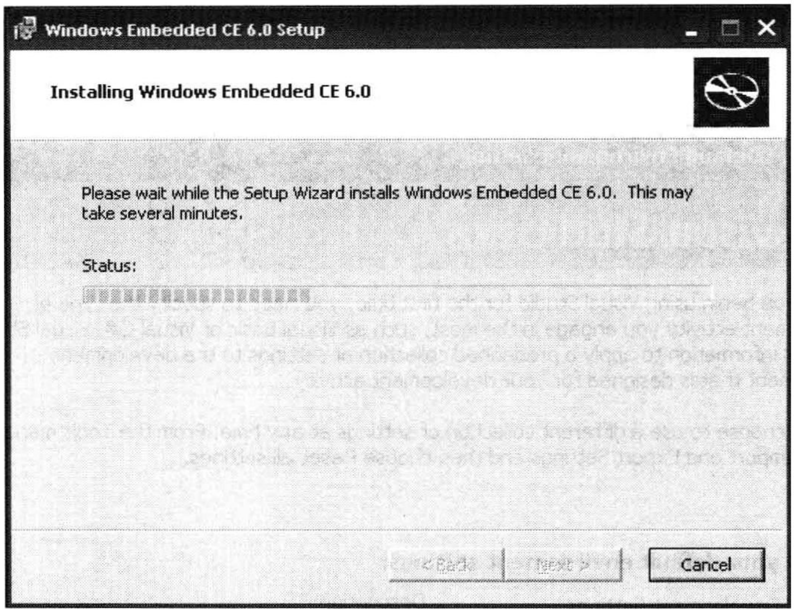



图2-18　安装进行中

当安装完成后，安装程序会提示已成功安装的对话框，单击“Finish”按钮结束安装程序，如图2-19所示。

图2-19　安装结束

当Visual Studio 2005及Windows Embedded CE 6.0都安装好后，若尚未启动过Visual Studio 2005，则在第一次启动Visual Studio 2005时，程序会要求选择一个环境设定，此时请选择“Platform Builder Development Settings”，然后单击“Start Visual Studio”按钮，这时

Visual Studio 2005需要数分钟进行环境参数设定，接着便可以启动Visual Studio 2005了，如图2-20所示。

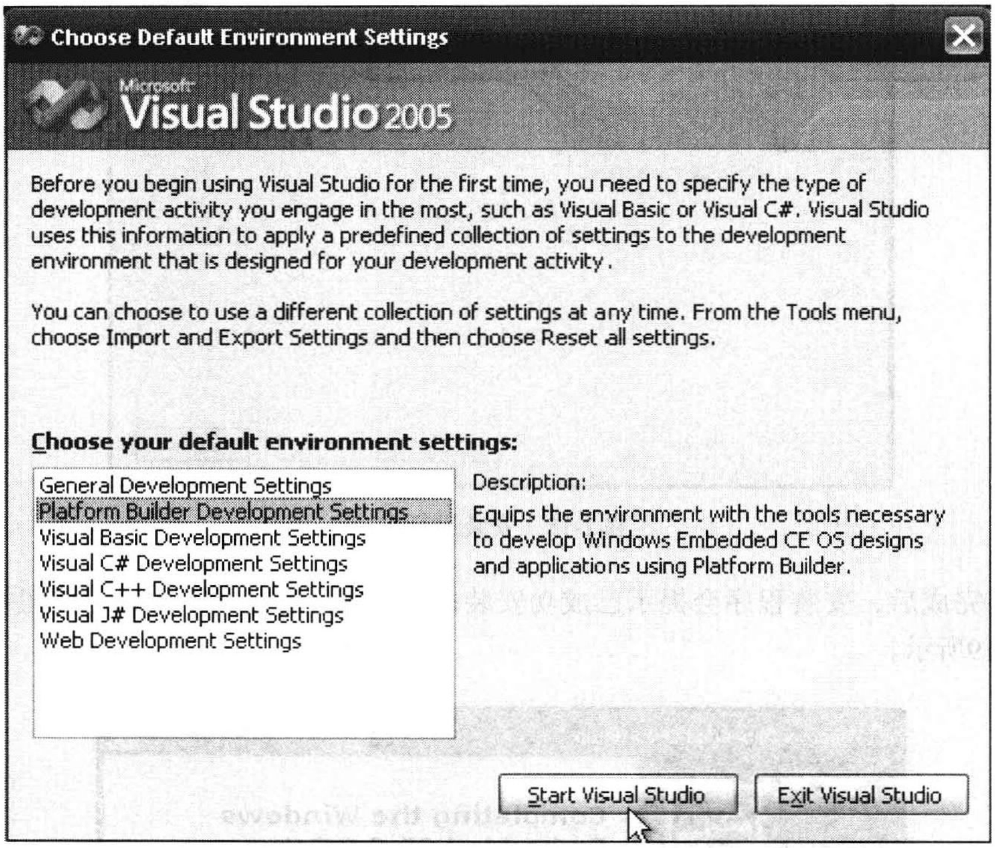



图2-20 选择默认环境设置

若曾经启动过Visual Studio 2005，则安装完Windows Embedded CE 6.0后，第一次启动Visual Studio 2005时，会出现如图2-21所示的对话框，提示用户最近已安装了Platform Builder，并询问是否重设Visual Studio 2005默认的环境设定，使其采用“Platform Builder Development Settings”为默认的环境设定。此时请单击“YES”按钮重设Visual Studio 2005的环境设定。

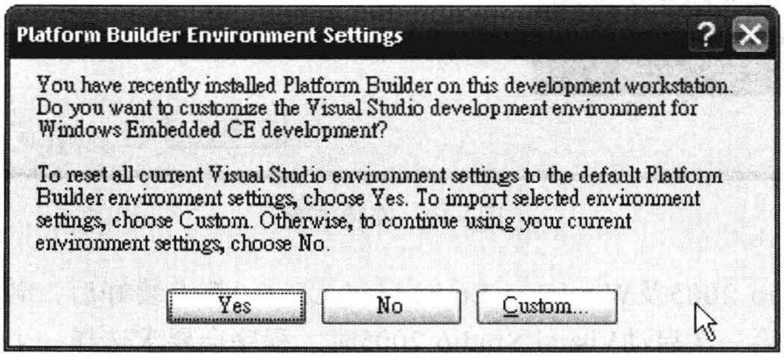



图2-21 重设Visual Studio 2005的环境设定

当将Visual Studio 2005重设环境设定为“Platform Builder Development Settings”后，则Visual Studio 2005主程序画面如图2-22所示。

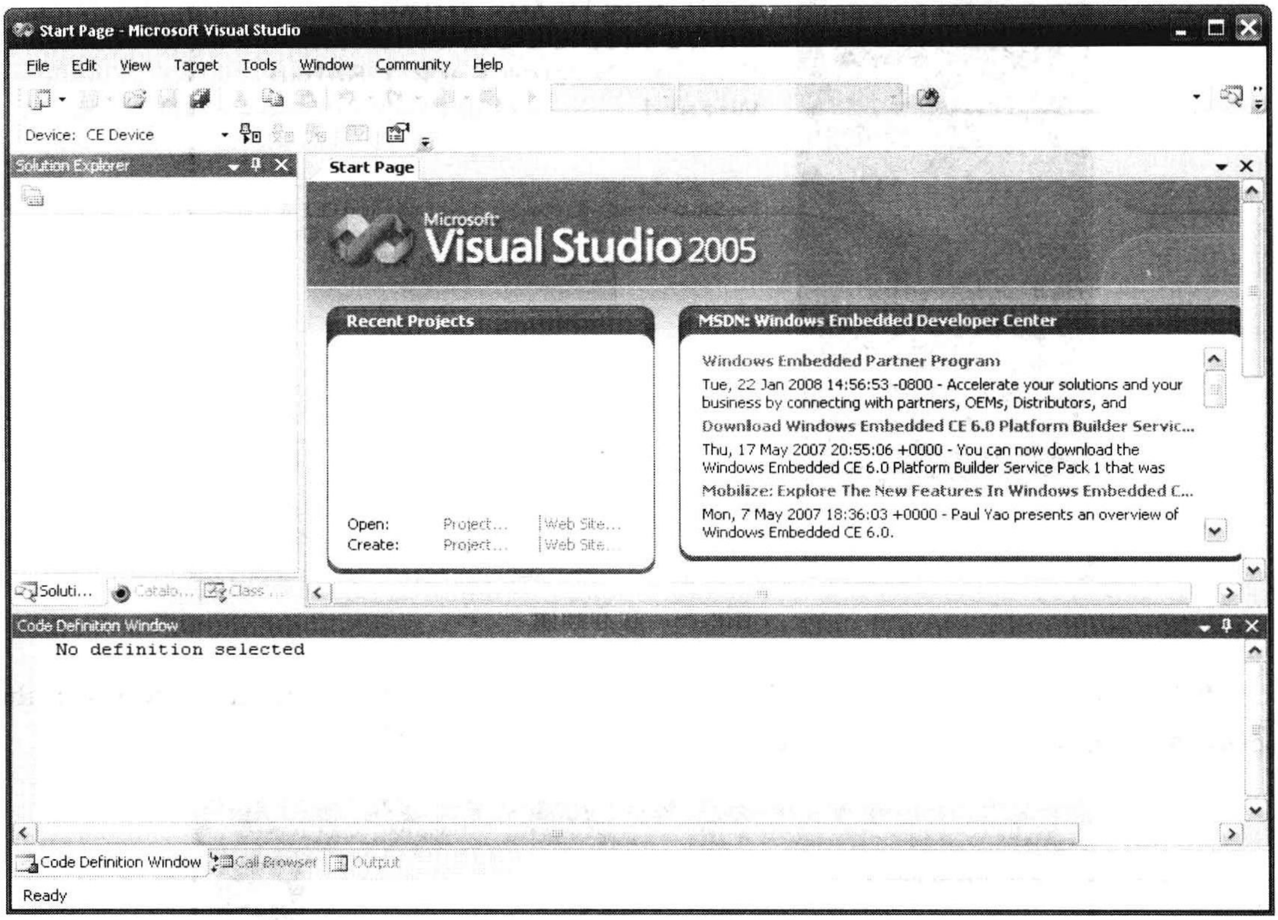



图2-22 Visual Studio 2005主程序画面

## 2.2.1 安装Platform Builder 6.0 Service Pack 1

微软公司在2007年4月发布了Platform Builder 6.0 Service Pack 1，为开发工具加入了几项新的特性，例如Remote Tools Framework、Hardware-Assisted Debugging等。用户可以在微软下载中心找到安装程序，或直接从http://www.microsoft.com/downloads/details.aspx?FamilyId=BF0DC0E2-8575-4860-A8E2-290ADF242678&displaylang=en#filelist这个页面下载安装程序，所下载的文件为“Windows Embedded CE 6.0 Platform Builder Service Pack 1.msi”。

执行下载的“Windows Embedded CE 6.0 Platform Builder Service Pack 1.msi”，安装程序首先会显示如图2-23所示的欢迎画面。

图2-23 欢迎画面

单击“Next”按钮后，会显示安装许可协议对话框，请选择“I accept the terms in the License Agreement”，然后单击“Next”按钮继续安装，如图2-24所示。

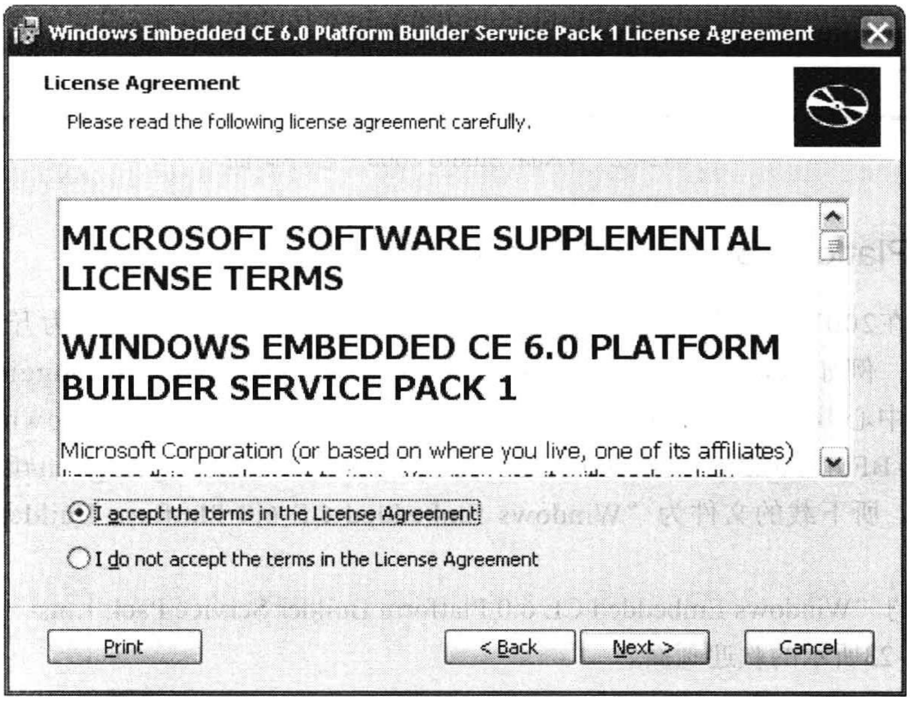



图2-24 安装许可协议

此时会显示如图2-25所示的对话框，单击“Install”按钮开始安装Platform Builder 6.0 Service Pack 1。

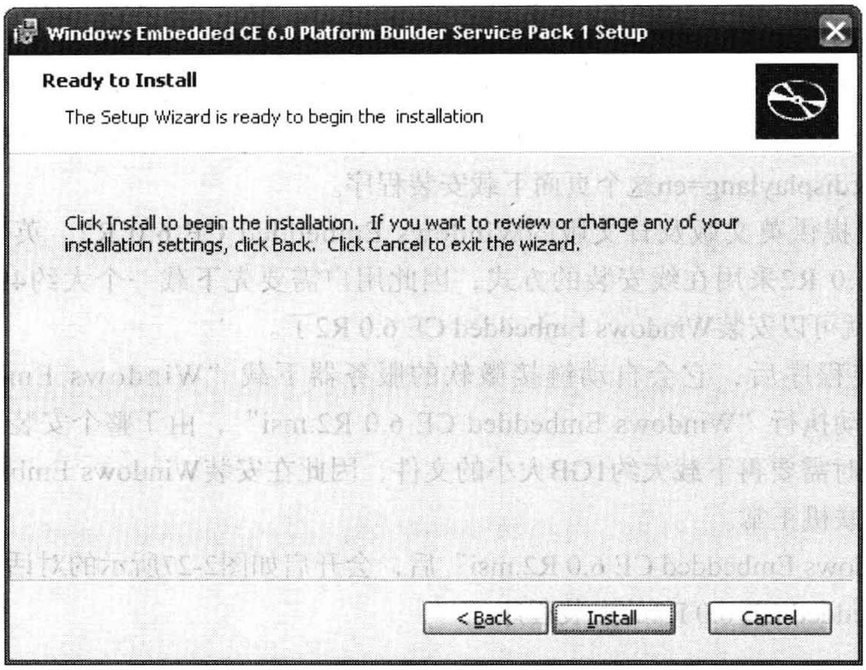



图2-25　开始安装Platform Builder 6.0 Service Pack 1

成功安装后会显示如图2-26所示的对话框，然后单击“Finish”按钮关闭安装程序即可。

图2-26　成功安装后关闭安装程序

## 2.2.2 安装Windows Embedded CE 6.0 R2

微软在2007年11月发布了Windows Embedded CE 6.0 R2，产品名称中的R2表示2nd Release的意思。Windows Embedded CE 6.0 R2中加入了更多新的特性，例如Video over IP telephony calls、Additional Voice over IP（VoIP）等。用户可以在微软下载中心获得安装程序，或直接由http://www.microsoft. com/downloads/details.aspx?familyid=F41FC7C2-F0F2-4FD6-9366-B61E0AB59565&displaylang=en这个页面下载安装程序。

微软目前仅提供英文版及日文版的Windows Embedded CE 6.0 R2。英文版的Windows Embedded CE 6.0 R2采用在线安装的方式，因此用户需要先下载一个大约49KB的安装程序（setup.exe），就可以安装Windows Embedded CE 6.0 R2了。

当执行安装程序后，它会自动链接微软的服务器下载“Windows Embedded CE 6.0 R2.msi”，并自动执行“Windows Embedded CE 6.0 R2.msi”，由于整个安装过程采用在线安装，且完整安装时需要再下载大约1GB大小的文件，因此在安装Windows Embedded CE 6.0 R2时，请确保网络联机正常。

执行“Windows Embedded CE 6.0 R2.msi”后，会开启如图2-27所示的对话框，提示正在执行Windows Embedded CE 6.0 R2的安装程序。

图2-27 Windows Embedded CE 6.0 R2.msi安装向导

单击“Next”按钮后，会显示安装许可协议，请仔细阅读微软的授权条款，然后选择“I accept the terms in the License Agreement”，单击“Next”按钮继续安装，如图2-28所示。

因为Windows Embedded CE 6.0 R2增加了数个BSP支持，所以安装程序会提示用户选择需要的BSP，此处通常采用安装程序默认的选项，也可以全部安装，当然这会增加需要下载的文

件容量。如图2-29所示。

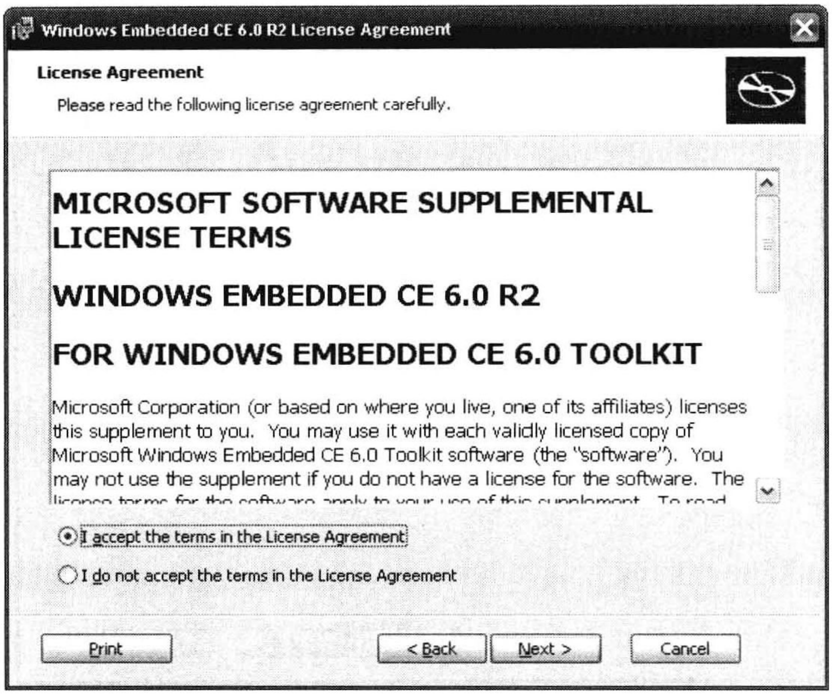



图2-28　客户安装许可协议

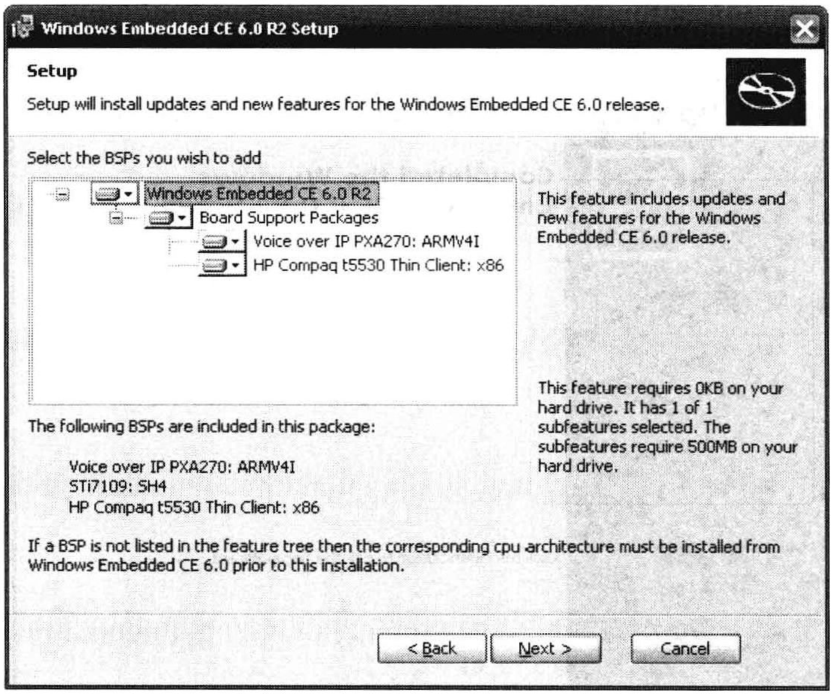



图2-29　选择想要安装的BSP

单击“Next”按钮后，会显示如图2-30所示的对话框，表示即将开始安装Windows

Embedded CE 6.0 R2，单击“Install”按钮开始安装。

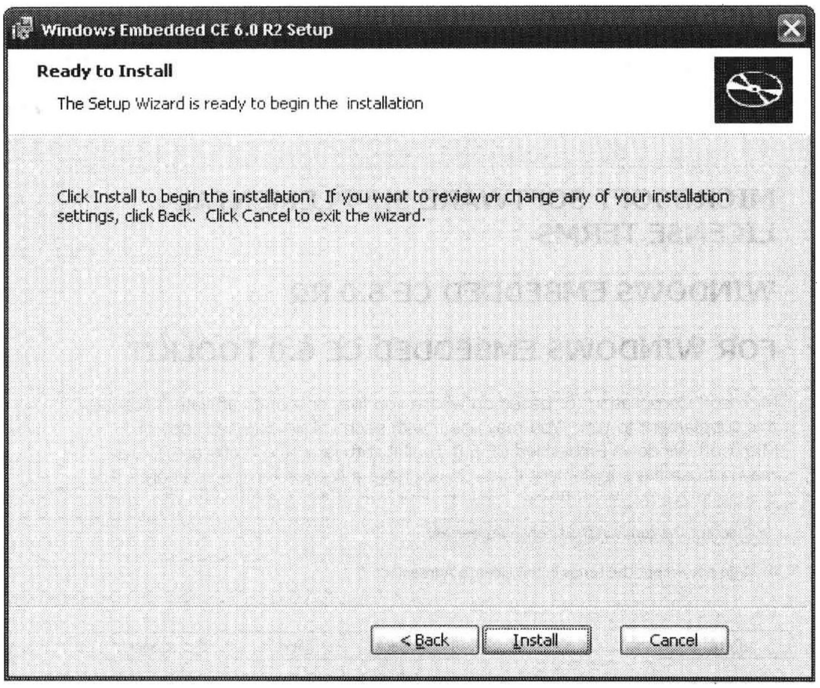



图2-30 开始安装

由于整个安装过程采用在线安装，且需要从微软的服务器下载大量的文件，因此请耐心等待安装完成，安装完成后如图2-31所示。

图2-31 安装完成

到这里Windows Embedded CE 6.0 R2的开发环境基本上已经建立好了，可以准备进入Windows Embedded CE 6.0系统及应用程序的开发世界了。此外，微软也一直在发布各项工具软件的补丁程序，这些补丁程序有些可能会增加功能支持，也有些会修正程序的Bug，因此建议用户花时间留意微软发布的补丁程序。

### 2.2.3　安装Windows Embedded CE 6.0 R3

微软在2009年8月发布了Windows Embedded CE 6.0 R3，产品名称中的R3表示3nd Release的意思。Windows Embedded CE 6.0 R3中加入了更多新的特性，例如Silverlight for Windows Embedded、Connection Manager、Touch Gestures、Third Party Applications（Adobe Flash Lite 3.0、Westtek File Viewers和Tencent Mobile QQ）等。用户可以在微软下载中心获得安装程序，或直接从http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=bc247d88-ddb6-4d4a-a595-8eee3556fe46这个页面下载安装程序。下载的文件为“Windows Embedded CE 6.0 R3.msi”。

执行“Windows Embedded CE 6.0 R3.msi”后会打开如图2-32所示的对话框，提示正在执行Windows Embedded CE 6.0 R3的安装程序。单击“Next”按钮。

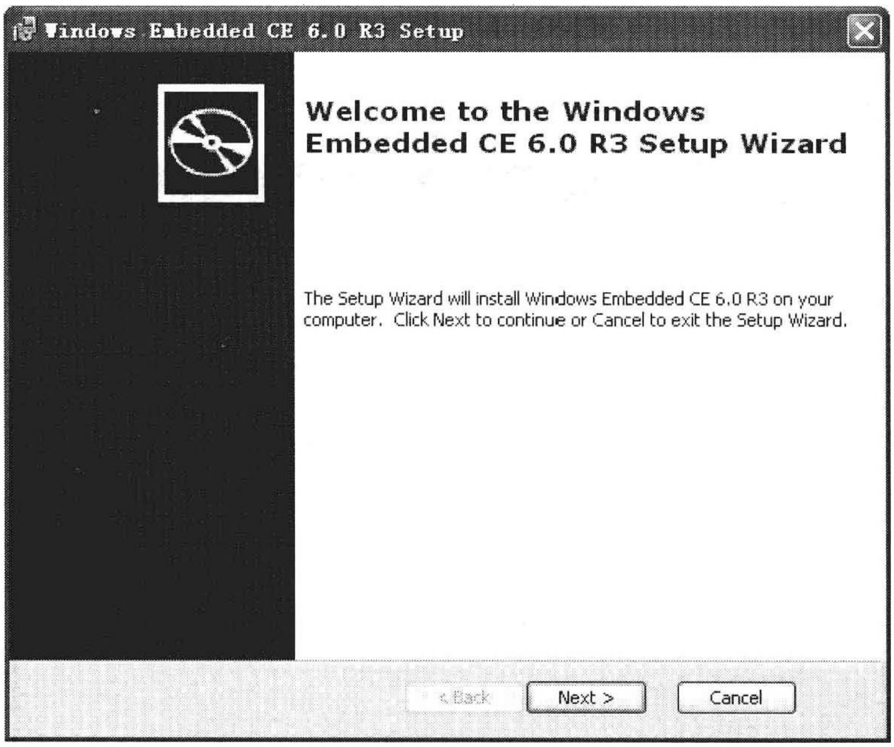



图2-32　Windows Embedded CE 6.0 R3安装向导

在安装许可协议中单击“I accept the terms in the License Agreement”，然后单击“Next”按钮继续安装，如图2-33所示。

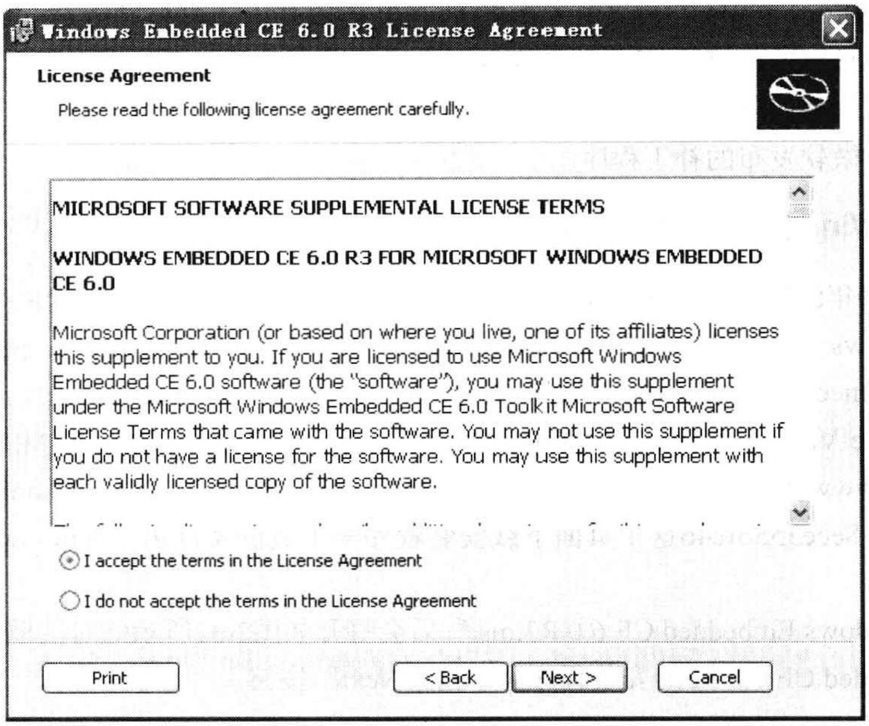



图2-33 客户安装许可协议

然后单击“Install”按钮，如图2-34所示。

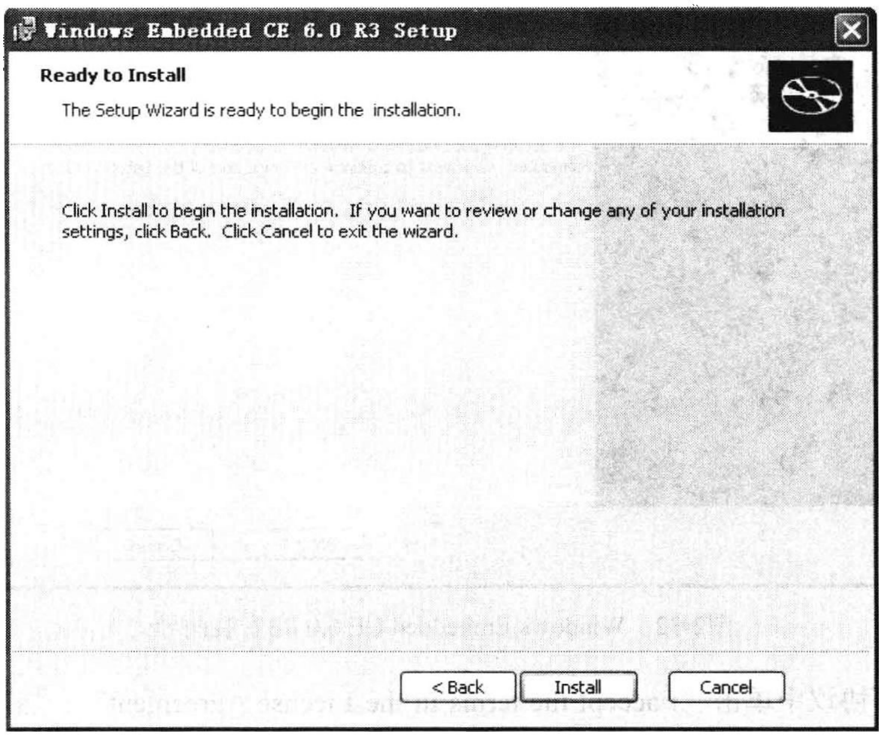



图2-34 准备安装

安装完成后单击“Finish”按钮结束安装，如图2-35所示。

图2-35　结束安装

Windows Embedded CE 6.0 R3的开发环境基本上已经建立好了，可以准备进入Windows Embedded CE 6.0系统及应用程序的开发世界了。

## 2.3　定制DMA-6410 XP系统

需要注意的是，本书介绍制作的系统是在DMA-6410XP实验平台上执行的，所以需要把DMA-6410XP的BSP开发包复制到Windows Embedded CE 6.0的安装目录的PLATFORM目录下。

### 2.3.1　使用项目向导

在安装完Visual Studio 2005和Platform Builder for CE 6.0后，找到获得的DMA-6410XP，然后打开光盘的Win CE6.0\BSP目录，解压缩，将解压得到S3C6410_SEC_V1.rar和DMA6410XP.rar两个文件。将S3C6410_SEC_ V1.rar这个文件复制到“\WIN CE600\PLATFORM\COMMON\SRC\SOC”目录下，并将其解压缩到此目录下，如图2-36和图2-37所示。

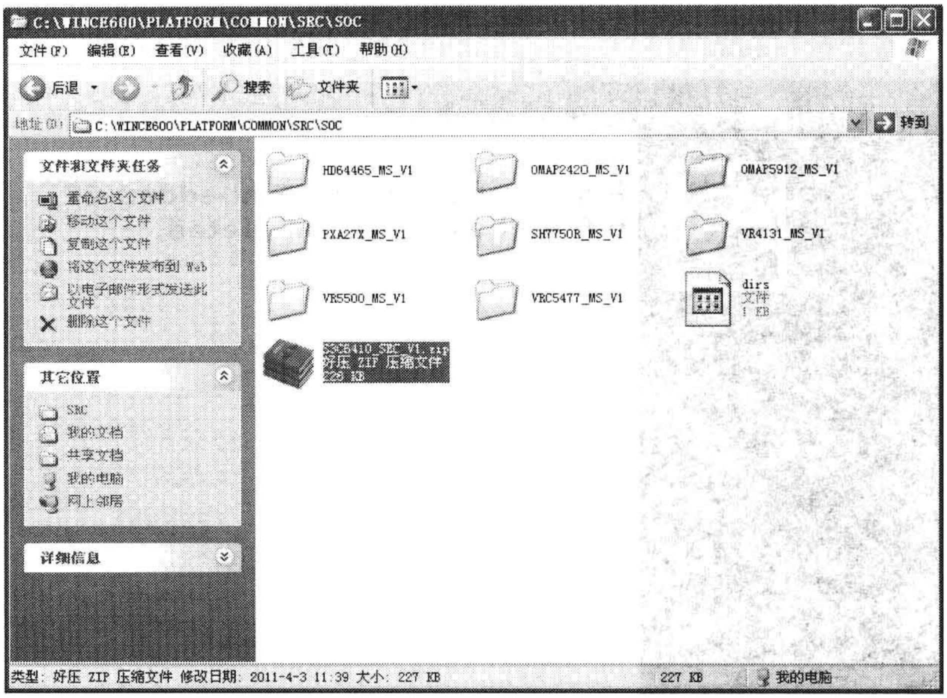



图2-36 复制S3C6410_SEC_V1

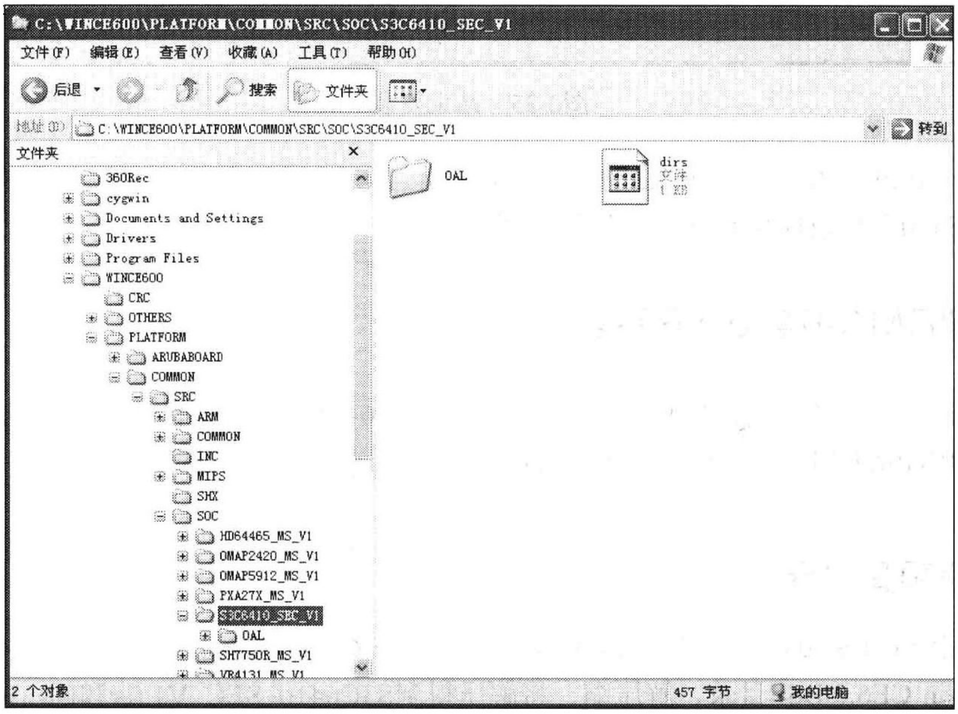



图2-37 解压缩S3C6410_SEC_V1

将DMA6410XP.rar这个文件复制到WinCE 6.0安装目录的"\WIN CE600\ PLATFORM"目录下，然后解压缩，如图2-38所示。

图2-38　解压缩BSP

确保DMA6410XP为PLATFORM下的一级目录，如图2-39所示。

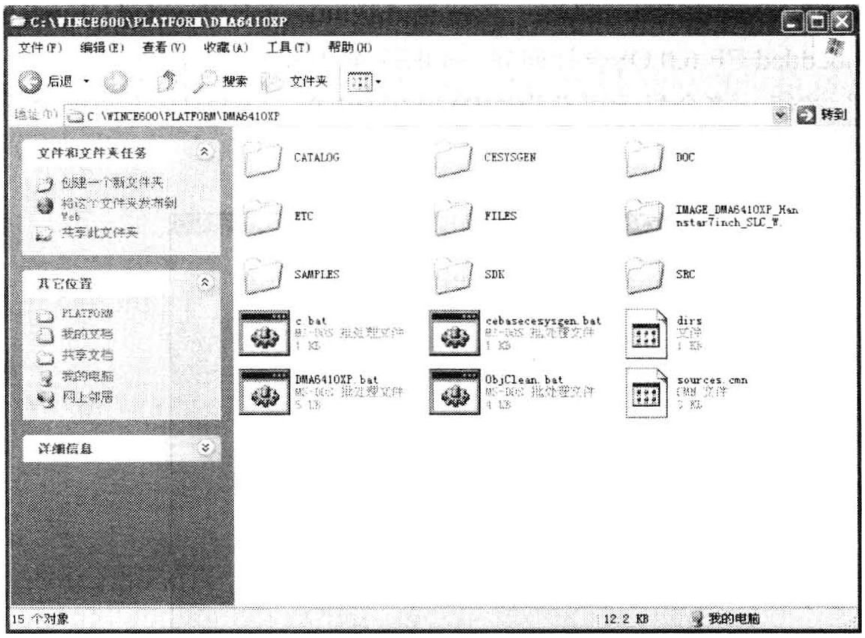



图2-39　解压后的BSP

## 2.3.2　构建新的平台

启动Visual Studio 2005，在菜单栏上依次单击“File”→“New”→“Project”，如图2-40所示。

图2-40 新建项目向导

选择Platform Builder for CE 6.0，输入“DMA6410XP”（可以是符合微软英文文件命名规则的任何字符），输入完毕后单击“OK”按钮进入Windows Embedded CE 6.0 OS设计向导。

Windows Embedded CE 6.0 OS设计向导，如图2-41所示。

单击“Next”按钮，将会显示可获得的DMA6410XP：ARMV4I（如果之前的BSP解压缩步骤操作正确），如图2-42所示。

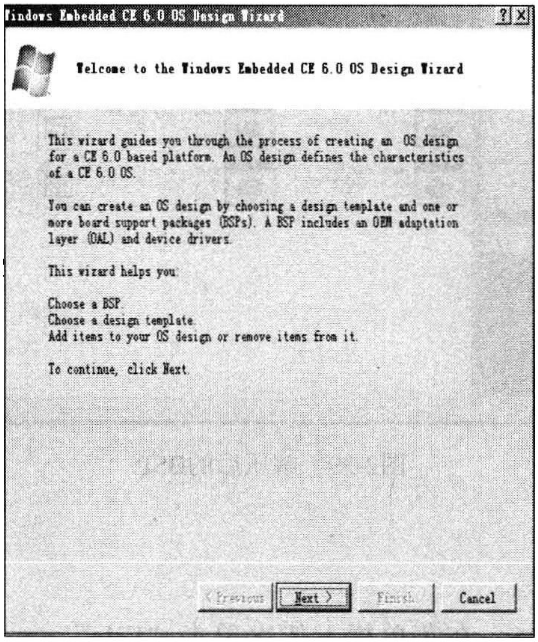



图2-41 向导的欢迎信息

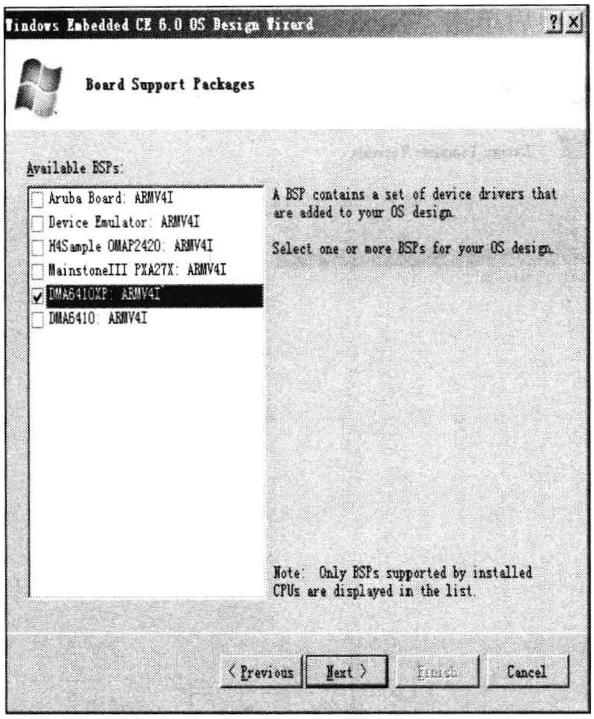



图2-42 在可用的BSP选项中选择DMA6410XP：ARMV4I

单击“Next”按钮，进入图2-43所示界面。

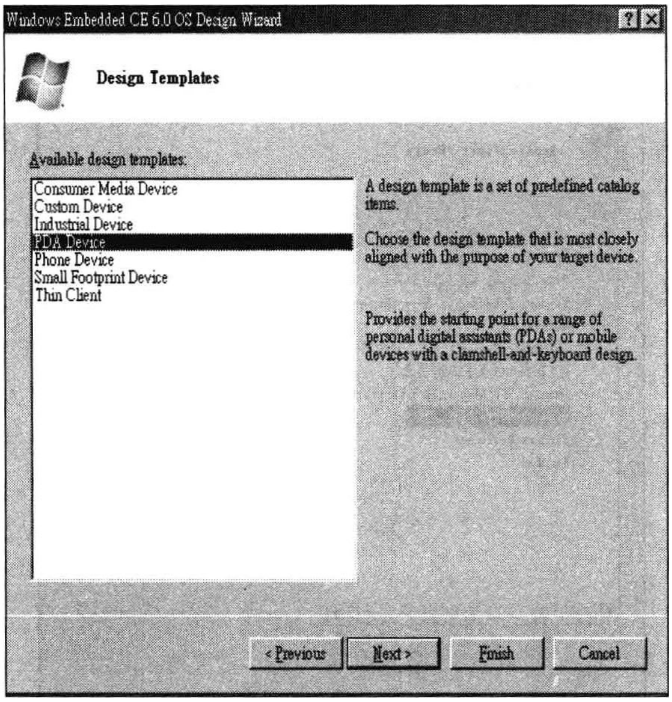



图2-43 选择设计模板一

单击“PDA Device”选项后单击“Next”按钮进入图2-44所示界面。

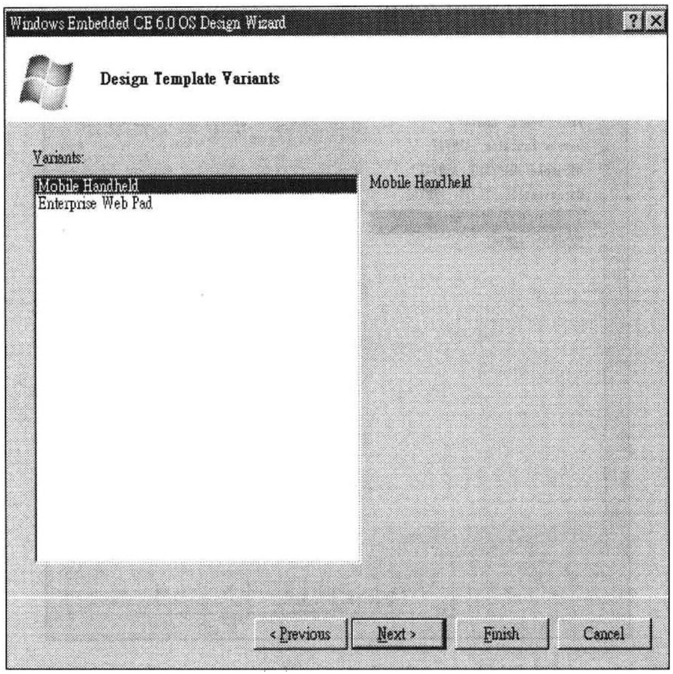



图2-44 选择设计模板二

单击“Next”按钮后进入应用程序和多媒体组件选择对话框，如图2-45所示，这里采用默认选项，然后单击“Next”按钮。

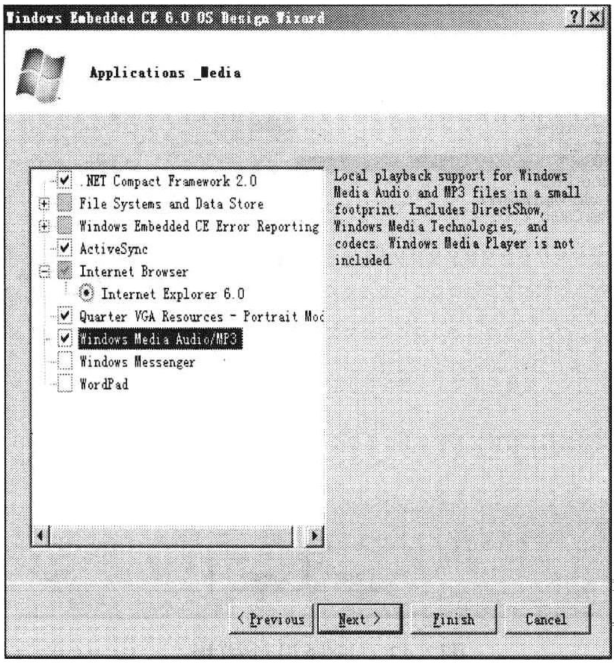



图2-45 应用程序和多媒体组件选择对话框

进入网络与通信组件选择对话框，如图2-46所示，这里采用默认选项，然后单击“Next”按钮。

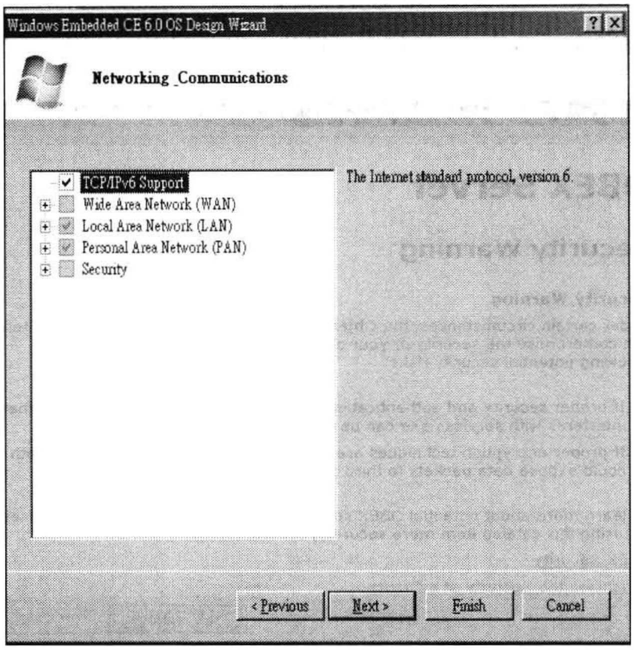



图2-46 网络与通信组件选择对话框

这样，Windows Embedded CE 6.0的创建工作就算是完成了，在如图2-47所示的对话框中单击“Finish”按钮关闭创建向导。

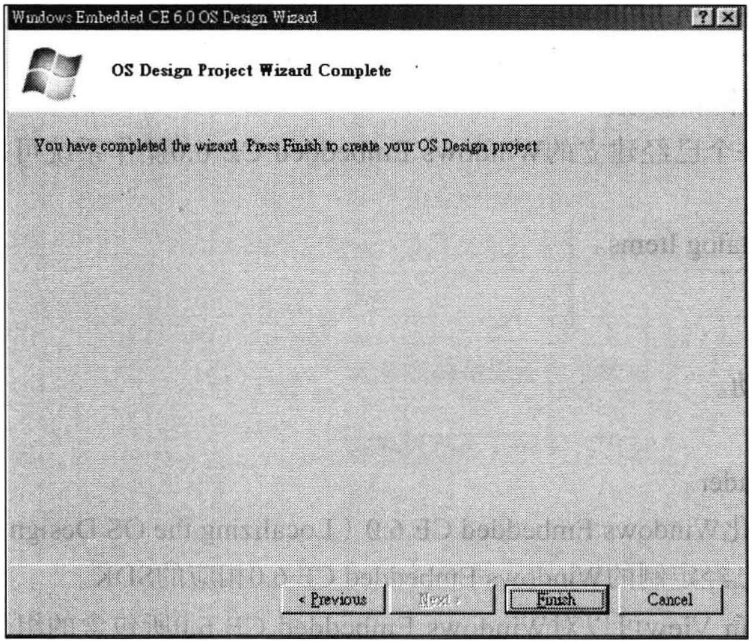



图2-47 创建完成

当单击“Finish”按钮时，根据所选组件的不同，可能会弹出一个安全警告（Security

Warning），如图2-48所示。由于模板PDA Device Enterprise Web Pad默认选择了OBEX和NDIS User-mode I/O Protocol，所以会弹出一个与之相关的警告窗口。如果确认在创建的Windows Embedded CE 6.0中加入了这些组件，请单击“Acknowledge”按钮，否则请单击“Cancel”按钮取消加入。

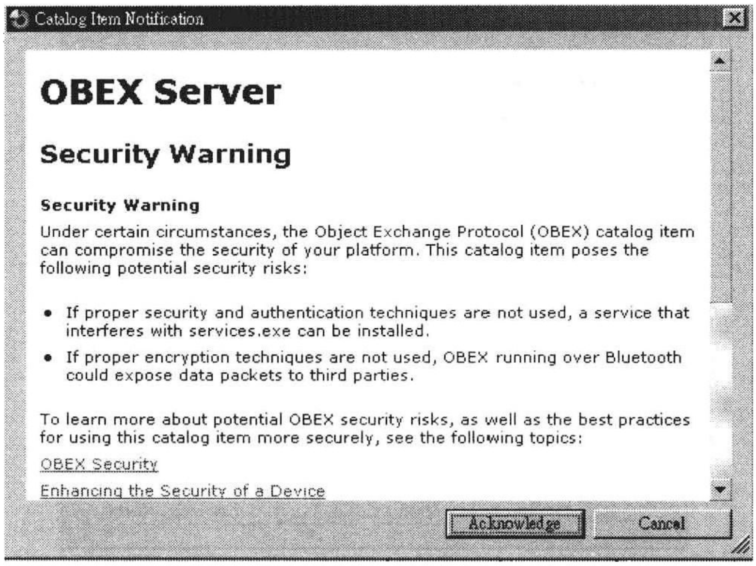



图2-48 安全警告

## 2.3.3 编辑和修改Windows Embedded CE项目

前面介绍了Windows Embedded CE 6.0操作系统的创建工作，在实际工作中，这只是操作系统开发中很小的一步，也是非常简单的一步。为了适应不同应用和不同设备的不同需求，不可避免地要进一步编辑和修改。

一般来说，对一个已经建立的Windows Embedded CE 6.0操作系统可以进行如下几个方面的编辑和修改操作：

- 添加/删除Catalog Items。
- 添加子项目。
- 添加BSP。
- 添加设备驱动。
- 建立OAL。
- 建立Boot Loader。
- 本地化/国际化Windows Embedded CE 6.0（Localizing the OS Design）。
- 配置和建立已经定制的Windows Embedded CE 6.0相应的SDK。

通过Catalog Item View可以对Windows Embedded CE 6.0所包含的组件进行删除和添加操作。在某个组件上单击鼠标右键，会弹出如图2-49所示的环境菜单，可以进行以下几种操作：

- 在新创建的Windows Embedded CE 6.0中添加/删除这个组件。
- 查看这个组件的依赖关系。

● 查看这个组件的属性。

组件和模块的添加一定要依据最小化的原则，能不添加的模块一律不添加，需要添加的模块也要考虑剔除其用得比较少的功能。可用通过查看一个模块的“Depends on”（依赖于哪些模块）和“Dependency of”（被哪些模块依赖）来进一步分析，如图2-50所示。

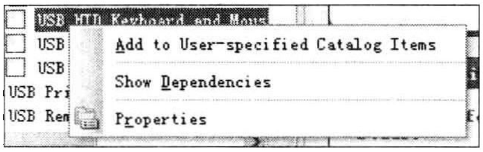



图2-49　组件的环境菜单

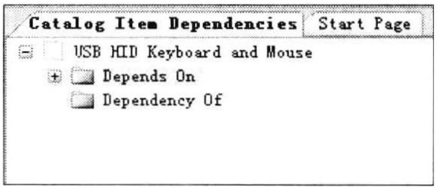



图2-50　查看组件的依赖关系

## 2.3.4　添加必要的功能

（1）添加“File Systems and Data Store”

展开Windows Embedded CE 6.0右边的Catalog列表，添加“File Systems and Data Store”，如图2-51所示。

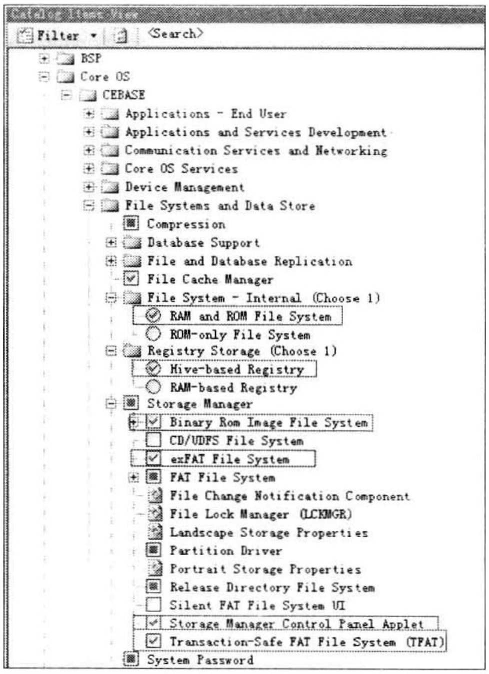



图2-51　添加“File Systems and Data Store”

（2）添加SD卡驱动

展开Windows Embedded CE 6.0右边的Catalog列表，在Catalog列表中单击“Device Drivers”→“SD”→“SD Bus Driver”和“Device Drivers”→“SDIO”→“SD Memory”，直接选择相应项目，将SD卡驱动组件加入当前平台，如图2-52和图2-53所示。

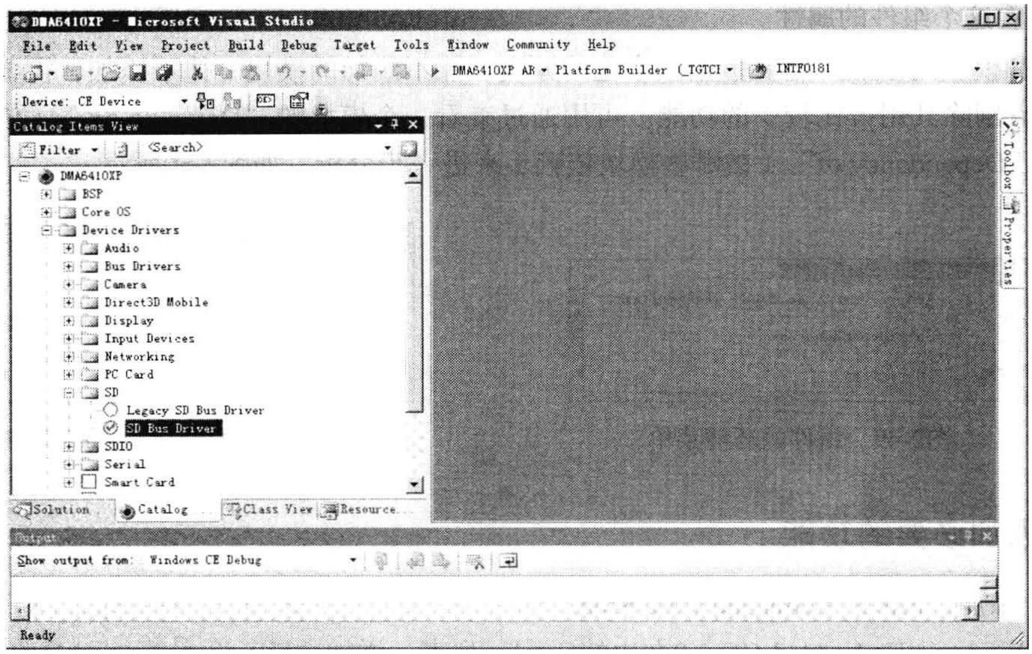



图2-52 添加SD卡驱动

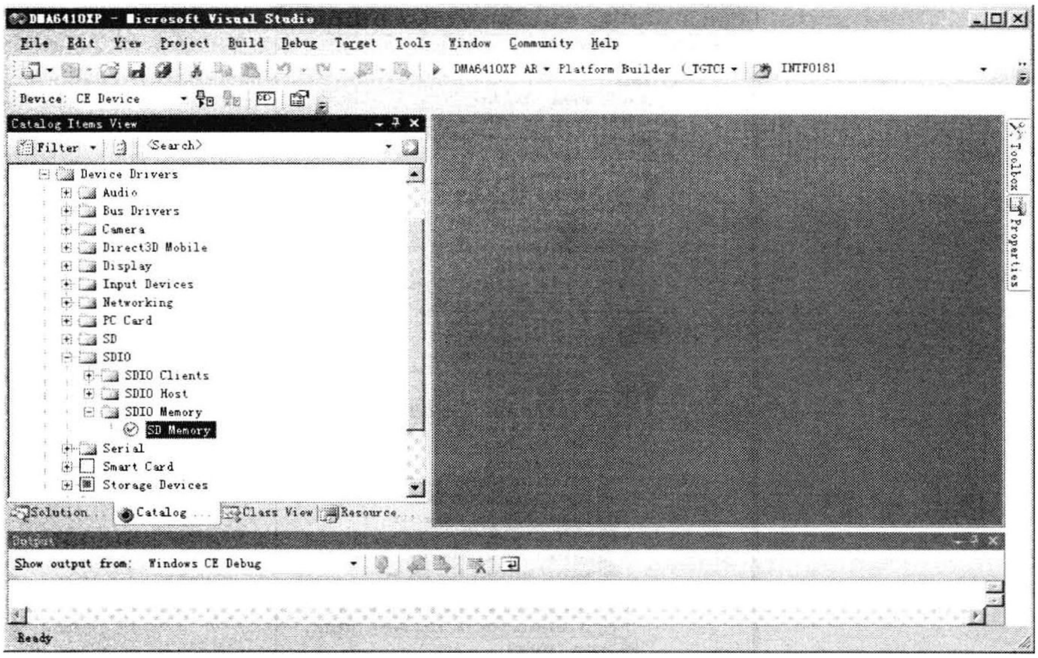



图2-53 添加成功

（3）添加鼠标和键盘驱动

单击Catalog列表中“Core OS”→“CE BASE”→“Core OS Services”→“USB Host Support”→“USB Human Input Device[HID] ClassDriver”→“USB HID Keyboard and Mouse”组件，将该组件加入到当前平台，如图2-54所示。

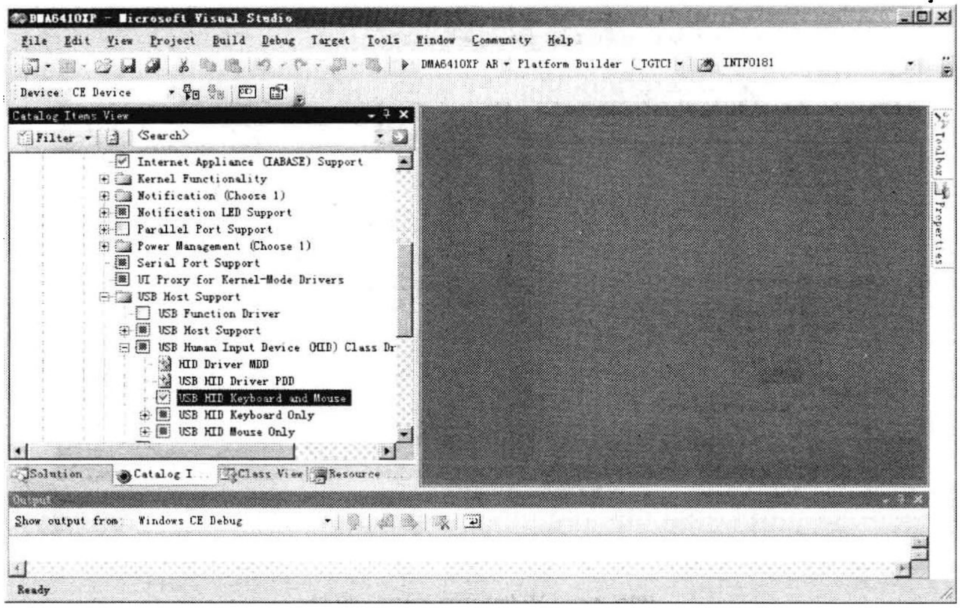



图2-54 添加鼠标和键盘驱动

（4）添加USB存储驱动

单击Catalog列表中“Core OS”→“CE BASE”→“Core OS Services”→“USB Host Support”→“USB Storage Class Driver”组件，将该组件加入到当前平台，如图2-55所示。

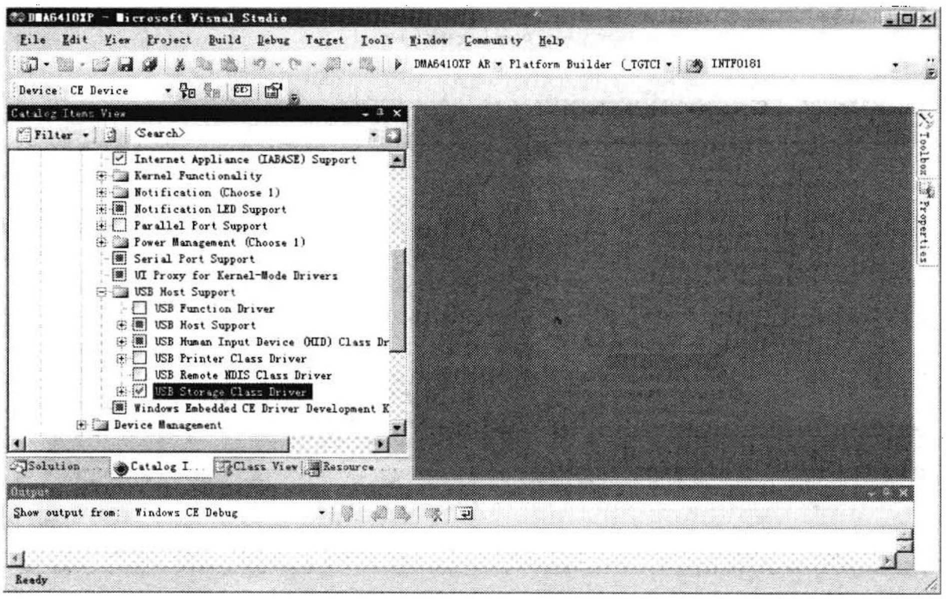



图2-55 添加USB存储驱动

（5）添加USB Client驱动

单击Catalog列表中“Device Drivers”→“USB Function”→“USB Function Clients”→“Serial”组件，将该组件加入到当前平台，如图2-56所示。

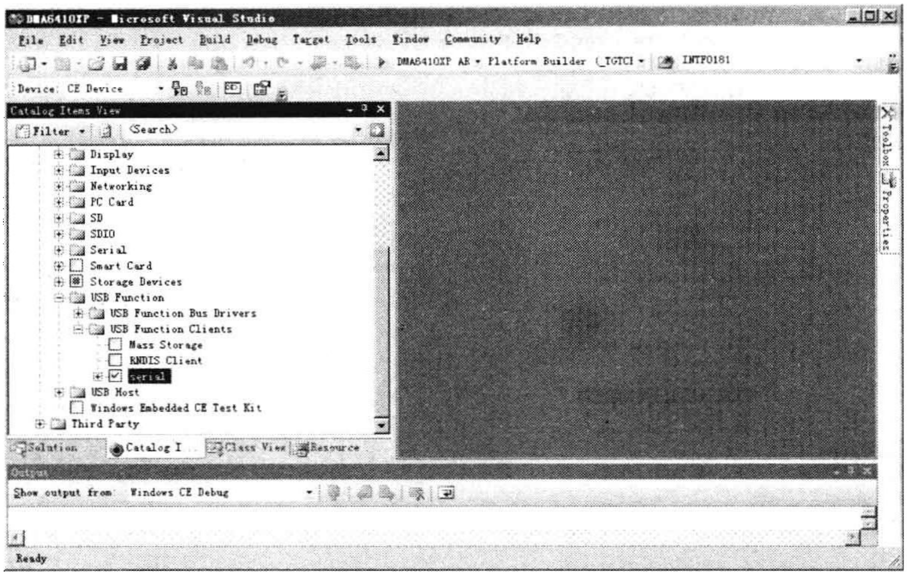


图2-56 添加USB Client驱动

（6）添加字型

单击Catalog列表中“Core OS”→“CE BASE”→“International”→“Locale Specific Support”→“Chinese（Traditional）”→“Fonts”→“MingLiU& PMingLiU（Choose 1）”→“MingLiU& P MingLiU”组件，将该组件加入到当前平台，如图2-57所示。

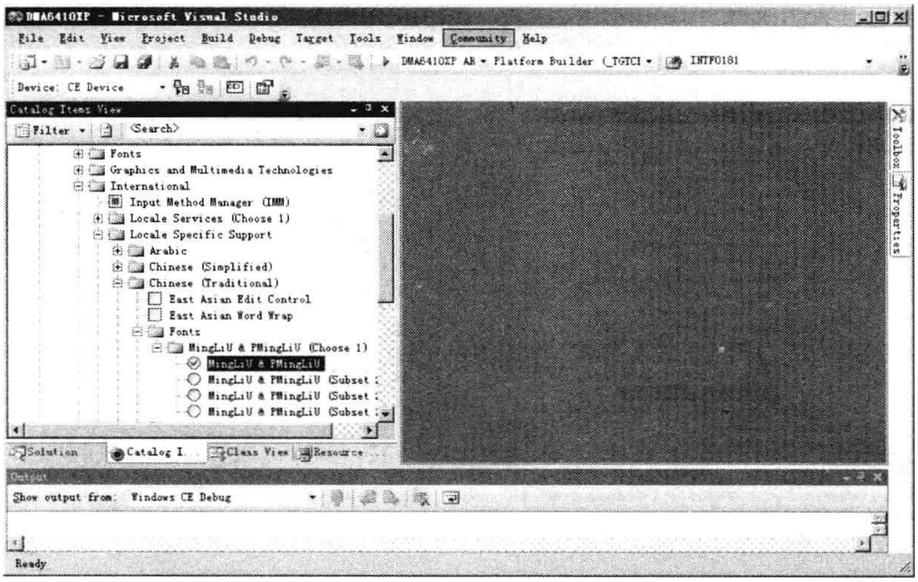


图2-57 添加字型

（7）添加IE浏览器

单击Catalog列表中“Core OS”→“CE BASE”→“International Client Services”→“Browser Application”→“Internet Explorer 6.0 Sample Browser”组件，将该组件加入到当前平台，如图2-58所示。

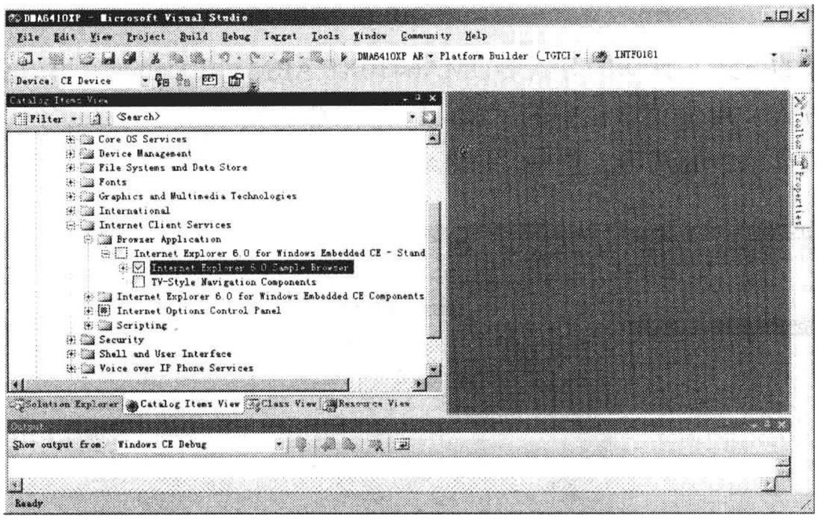


图2-58　添加IE浏览器

（8）添加“Graphics and Multimedia Technologies”组件

单击Catalog列表中“Core OS”→“CE BASE”→“Graphics and Multimedia Technologies”→“Graphics”栏目中相应组件，将这些组件加入到当前平台，如图2-59所示。

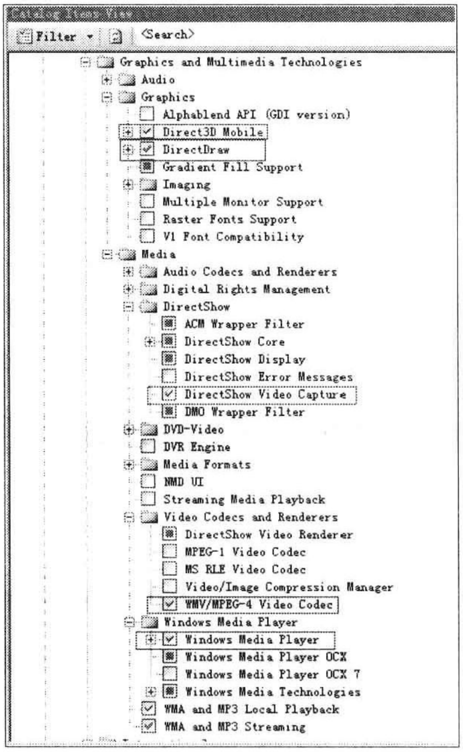


图2-59　添加“Graphics and Multimedia Technologies”组件

## 2.3.5 Build

Build之前要做的就是设置与Build相关的环境变量，检查与Build有关的设置，做好准备

后，就可以编译了。

首先更改编译的方式，默认的方式为DMA6410XP ARMV4I Debug，可以更改为DMA6410XP ARMV4I Release方式。单击菜单栏中的“Build”→“Configuration Manager”→“Active solution configuration”下拉菜单中的选项，选中“DMA6410 XP ARMV4I Release”，如图2-60所示。

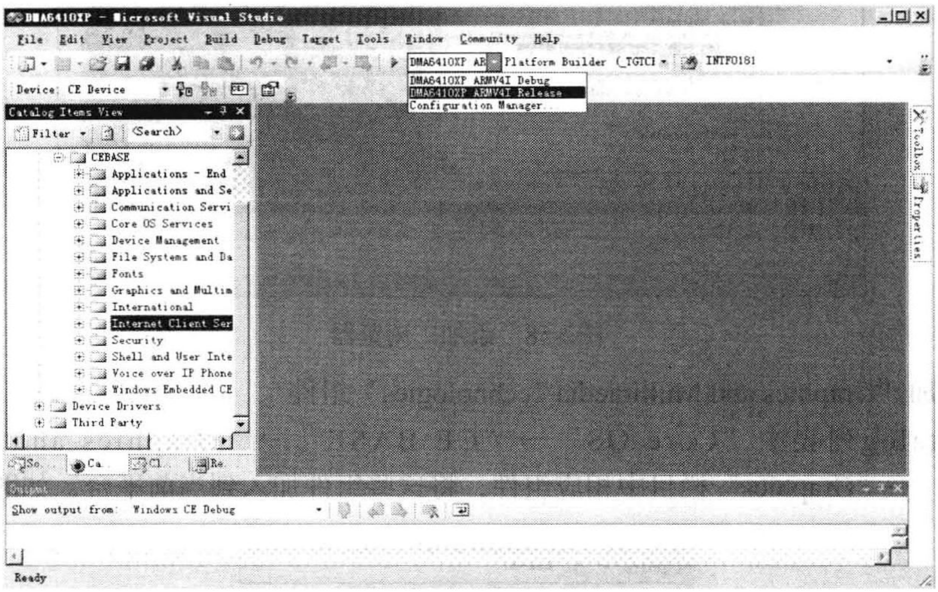


图2-60 更改编译模式

接下来设置系统的语言，在默认情况下，定制的操作系统的语言为英语，开发人员也可以根据本地化的需求，进行语言上的设置。单击菜单栏中的“Project”→“Properties”可以打开项目的属性页，也可以使用“Alt+F7”快捷键。在属性页中找到Locale列表框，先单击“Clear All”清除Platform Builder for CE 6.0的默认设置，然后在Locales中选择“中文（中国）”，在Default locale的下拉菜单中选择“中文（中国）”，系统的默认语言就被设置成了中文，如图2-61所示。

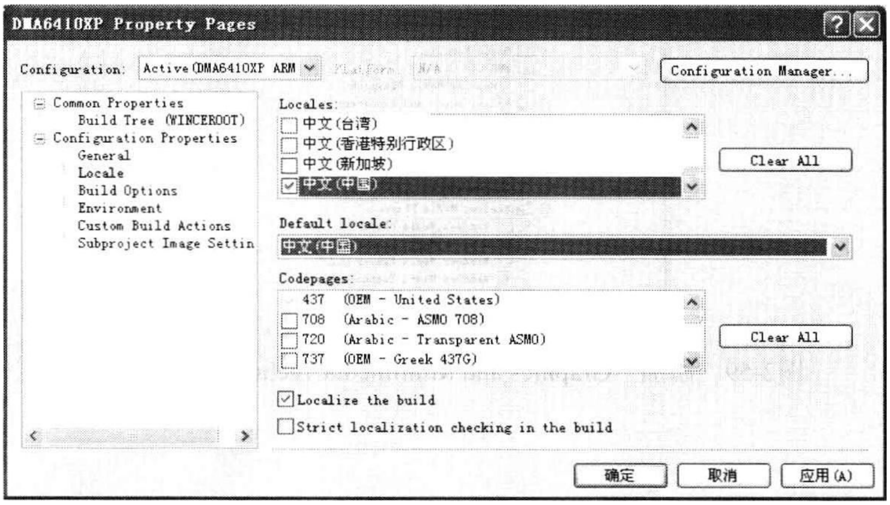


图2-61 DMA6410XP的属性页一

单击左侧列表框中的“Build Options”，在右侧取消“Enable KITL(no IMGNOKITL=1)”选择，如图2-62所示。

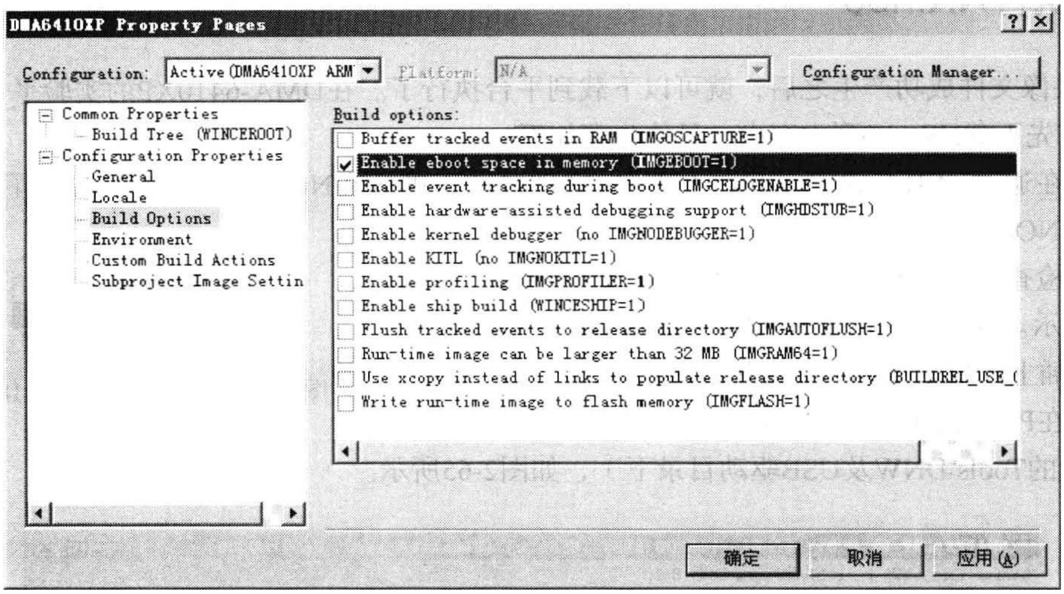


图2-62　DMA6410XP的属性页二

当确认所有的设置都完成并存储之后，就可以开始生成一个映像了。

在Visual Studio 2005中依次单击“Build”→“Advanced Build Commands”→“Sysgen”，开始生成系统。这是一个比较漫长的过程，一般需要15分钟以上的时间。当出现如图2-63所示的内容时表示系统已生成。

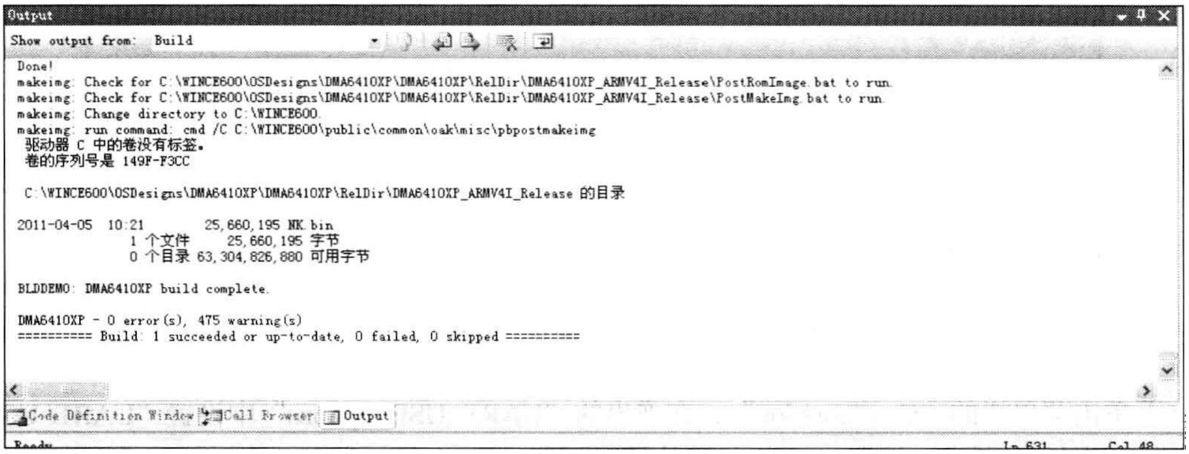


图2-63　映像生成完毕

一般情况下，在创建一个Windows Embedded CE 6.0项目时，最好能在WinCE600默认的项目目录内，因为在创建和Btuild的过程中，将会从WinCE600中复制很多文件，项目文件夹最后会有近1.5GB的大小。从这里也可以看出，要确认项目所在的磁盘至少有1.5GB的剩余空间，这也是为

什么在前面安装Windows Embedded CE 6.0时提到要求磁盘的剩余空间在10GB 以上。

## 2.4 执行NK.nb0

当映像文件成功产生之后，就可以下载到平台执行了。在DMA-6410XP的实验平台上执行系统，首先下载NK.nb0到内存中，具体步骤如下：

1）在通过USB下载Windows CE映像文件之前，确认在Nor Flash中已经烧写了“6410_OtgMon_NOR.bin”。

2）检查目标板SW118是否设置为“01010000”，如图2-64所示。配置BootLoader从Nor Flash启动。

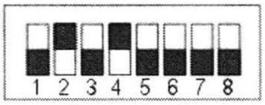



图2-64 检查目标板SW118的设置

3）插上随机附赠的5V/2A电源适配器。

4）在PC主机上运行“DNW.exe”（此文件位于随机附带光盘的Tools\DNW及USB驱动目录下），如图2-65所示。

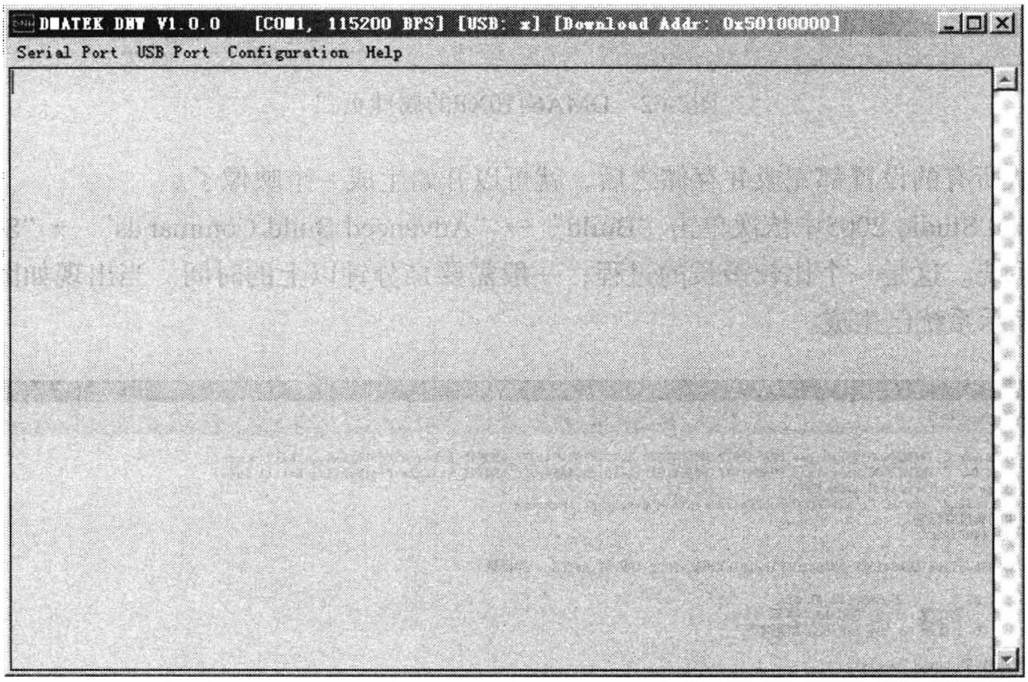



图2-65 在PC主机上运行“DNW.exe”

5）单击菜单栏的“Configuration”，在弹出的“UART/ USB Options”中设置“UART/ USB”的选项，如图2-66所示。根据PC主机的具体情况配置“Baud Rate”和“COM Port”（这里选择COM 1），在“Download Address”文本框中输入“0x50100000”，然后单击“OK”按钮。

6）用随机附带的串口线（一端是DBq，接到PC主机串行端口，另一端接到目标板的DEBUG，即UART0）将PC主机的串行端口和DMA-6410XP的调试串行端口DEBUG连接起来。然后单击菜单栏中的“Serial Port”→“Connect”，打开DMA-6410XP目标板的电源开关

“SW201”，并按下PC键盘上的Space（空格）键，出现如图2-67所示窗口。

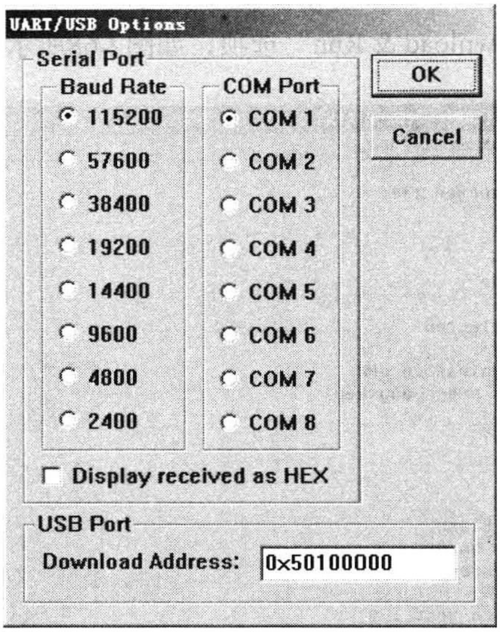



图2-66　设定串口参数

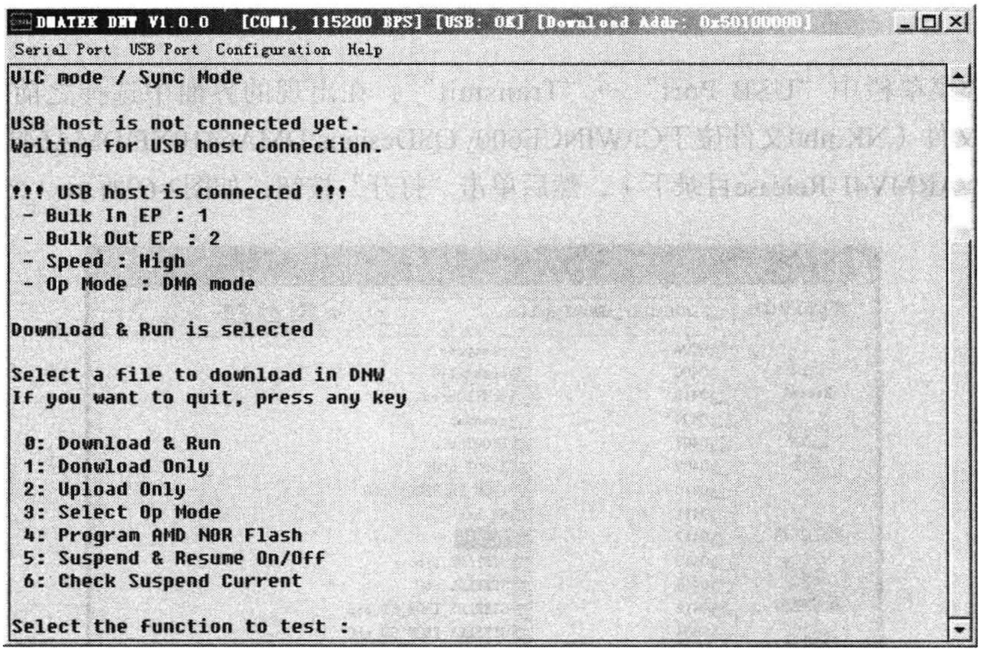



图2-67　连接成功

7）插入随机附赠的USB OTG线，将PC主机的USB接口和DMA-6410XP的USB-OTG接口连起来（即CON501）。若是第一次使用DMA-6410XP，还需要安装USB驱动（secusb2.sys和secusb2.inf位于随机附赠光盘Tools\DNW\SMDK6410 USB Driver目录下）；若已安装了此驱

动，则可以略过这一步。

8）插入后，出现“USB OTG Mon”的配置项，此时按下PC主机上的数字键“0”，并按下Enter键，表示选择“0：Download & Run”选项，如图2-68所示。

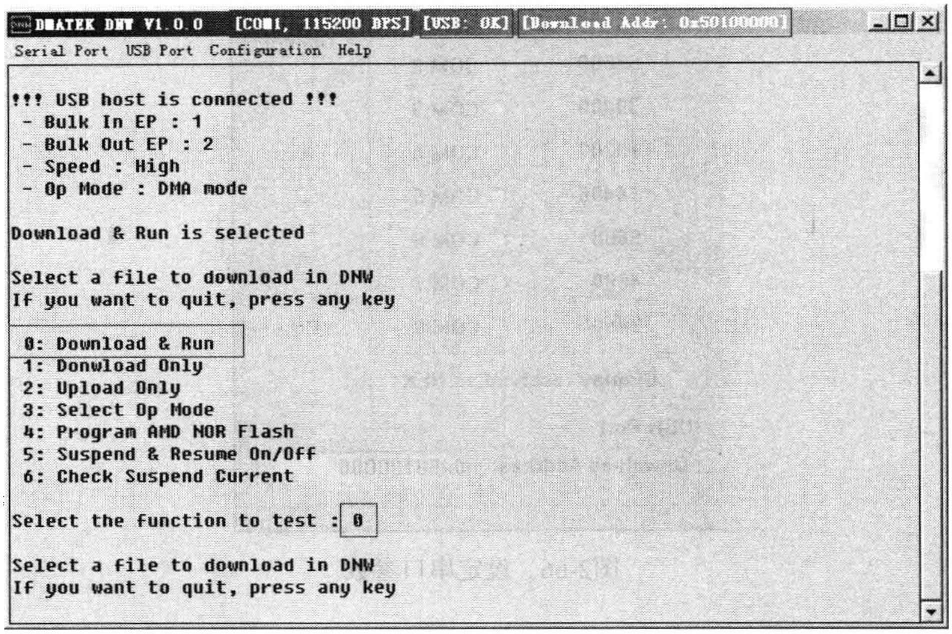



图2-68 下载并运行程序

9）单击菜单栏中“USB Port”→“Transmit”，在出现的界面中选择之前编译生成的“NK.nb0”文件（NK.nb0文件位于C:\WINCE600\ OSDesigns\DMA6410XP\DMA6410XP\RelDir\DMA6410XP_ARMV4I_Release目录下），然后单击“打开”按钮，如图2-69所示。

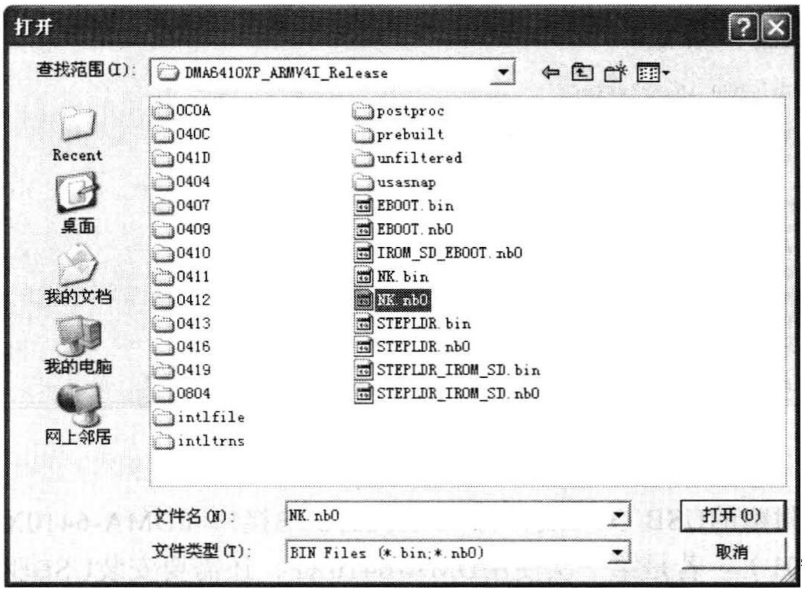



图2-69 选择NK.nb0文件

10）下载一开始，出现下载进度条。NK.nb0下载完毕后，Windows Embedded CE 6.0就可以在DMA-6410XP实验平台运行了，如图2-70所示。

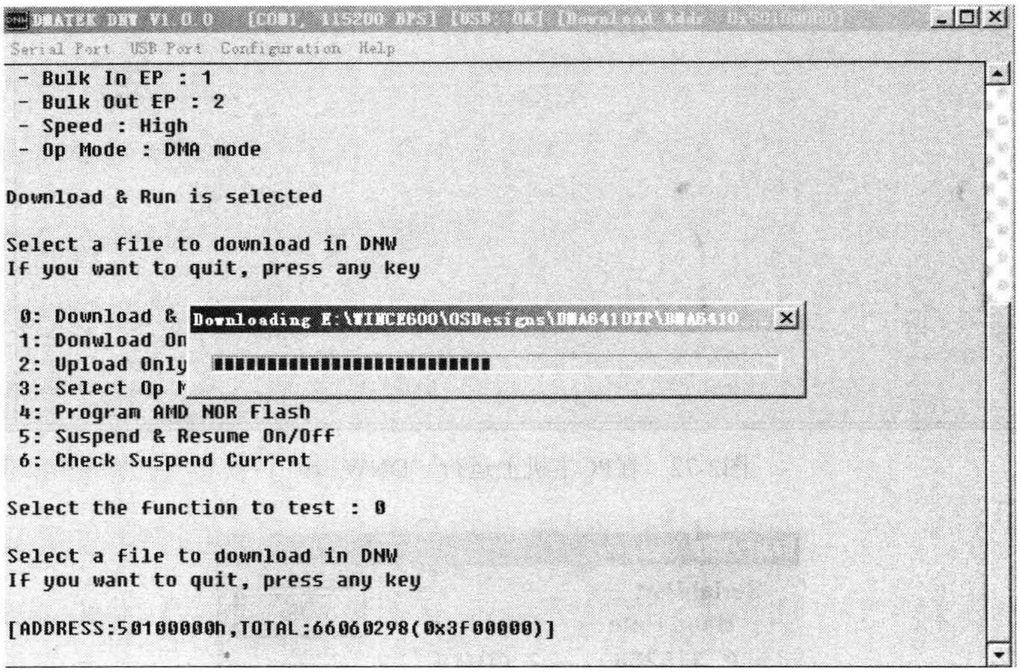



图2-70　运行Windows Embedded CE 6.0

## 2.5　Windows CE烧写

本书介绍如何把Windows CE烧写到实验平台上，并将其设置成自动启动。

可以通过USB烧录Windows CE映像文件至Nand Flash，以及设定自动启动Windows CE。烧录NK.bin至Nand Flash的流程：Nor Flash启动BootLoader→执行EBOOT.nb0→烧写STEPLDR.bin至Nand Flash→重新启动→执行EBOOT.nb0→烧写EBOOT.bin至Nand Flash→设定由Nand Flash启动Ethernet Boot Loader→烧写NK.bin至Nand Flash。

1）在通过USB烧录Windows CE映像文件之前，确认在Nor Flash中已经烧写了“6410_OtgMon_NOR.bin”。

2）检查目标板SW118是否设置成“01010000”，如图2-71所示。配置BootLoader从Nor Flash启动。

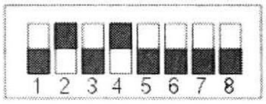



图2-71　检查目标板SW118的设置

3）插上随机附赠的5V/2A的电源适配器。

4）在PC主机上运行“DNW.exe”，如图2-72所示。

5）单击菜单栏中的“Configuration”，在弹出的“UART/USB Options”中设置“UART/USB”的选项，如图2-73所示。根据PC主机的具体情况配置“Baud Rate”和“COM Port”，（这里选择COM1），在“Download Address”文本框中输入“0x50030000”，然后单击“OK”按钮。

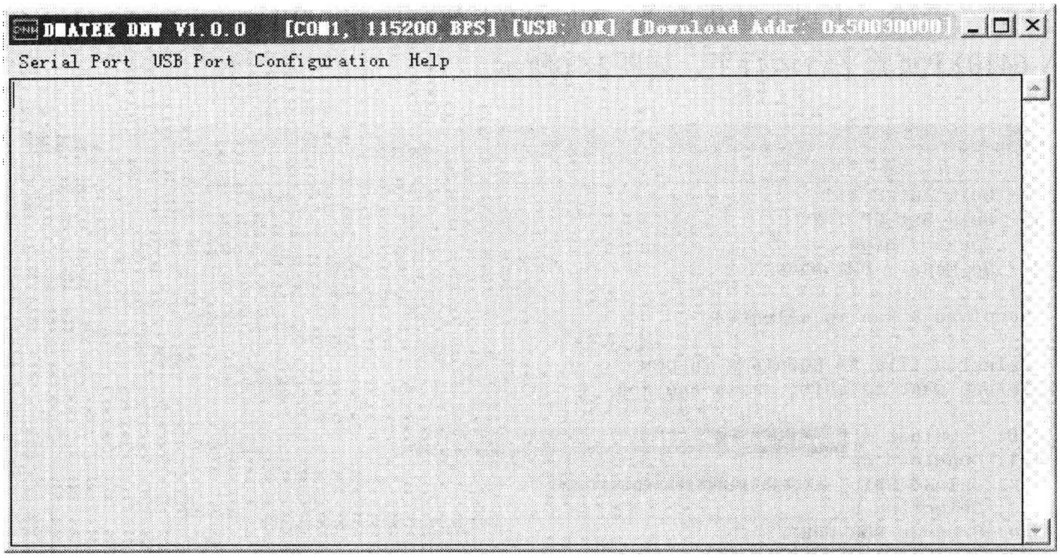



图2-72 在PC主机上运行“DNW.exe”

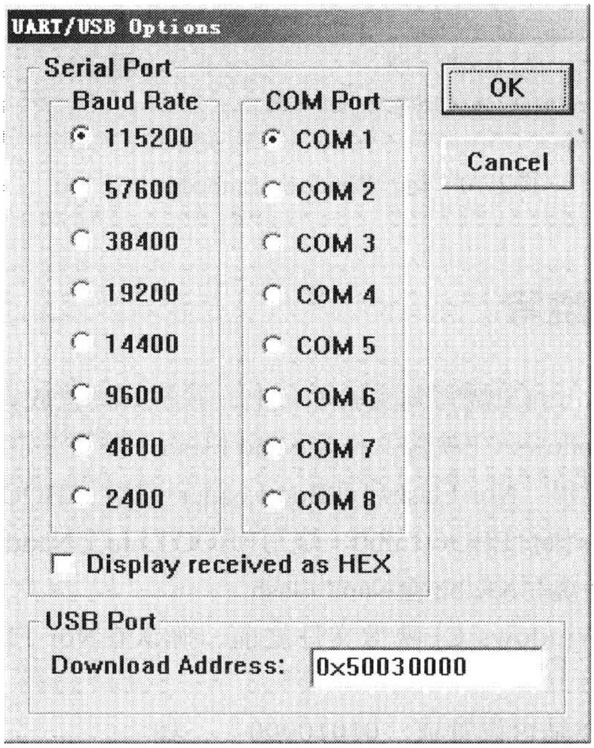



图2-73 设定串口参数

6）用随机附赠的串口线（一端是DB9，接到PC主机串行端口，另一端接到目标板的DEBUG，即UART0）将PC主机的串行端口和DMA-6410XP的调试串行端口DEBUG连接起来。然后单击菜单栏中的“Serial Port”→“Connect”，打开DMA-6410XP目标板的电源开关“SW201”，并按下PC主机上的SPACE（空格）键，出现如图2-74所示窗口。

7）用随机附赠的USB OTG线将PC主机的USB接口和DMA-6410XP的USB_OTG接口连接起来（即CON501），步骤参见上一节的步骤7。

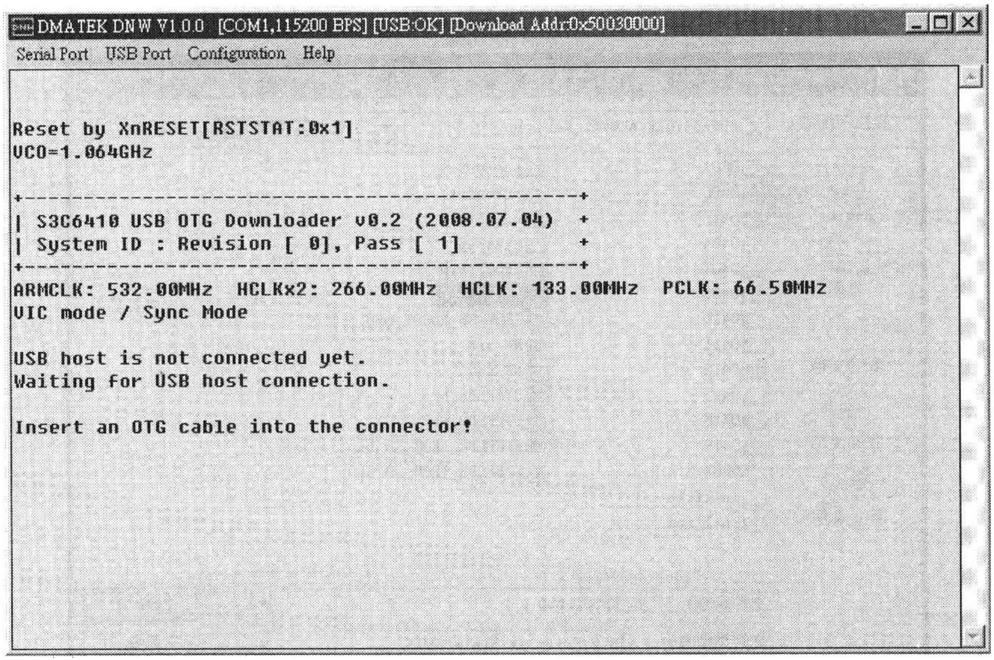



图2-74　连接成功

8）插入后，出现“S3C6410 USB OTG Downloader”的配置项，此时按下PC主机上的数字键“0”，并按下Enter键，表示选中“0：Download & Run”选项，如图2-75所示。

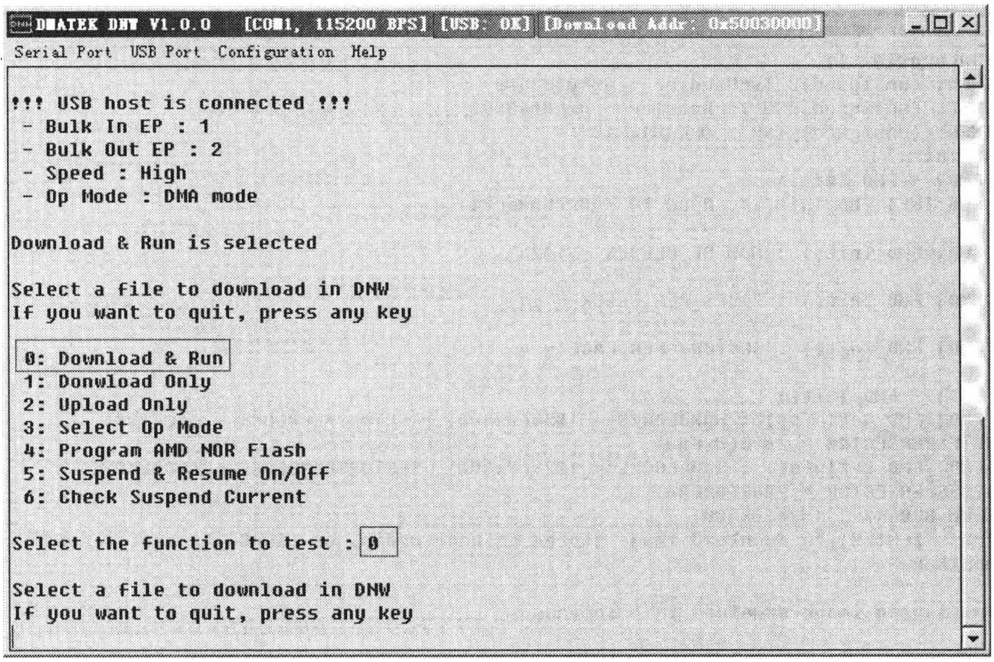



图2-75　下载并运行程序

9）单击菜单栏中“USB Port”→“Transmit”，在出现的界面中选择之前编译生成的“EBOOT.nb0”文件（EBOOT.nb0文件位于\ WIN CE600\ OSDesigns\DMA6410XP\DMA6410XP\RelDir\DMA6410XP_ARMV4I_Release目录下），然后单击“打开”按钮，如图2-76所示。

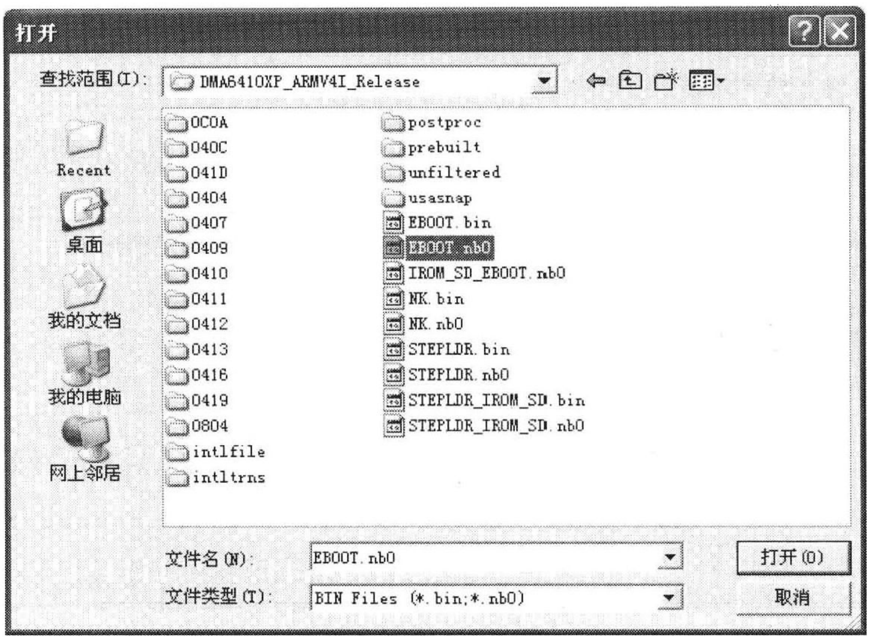



图2-76 选择EBOOT.nb0文件

10）在EBOOT.nb0下载完毕后，将会出现如图2-77所示的窗口。

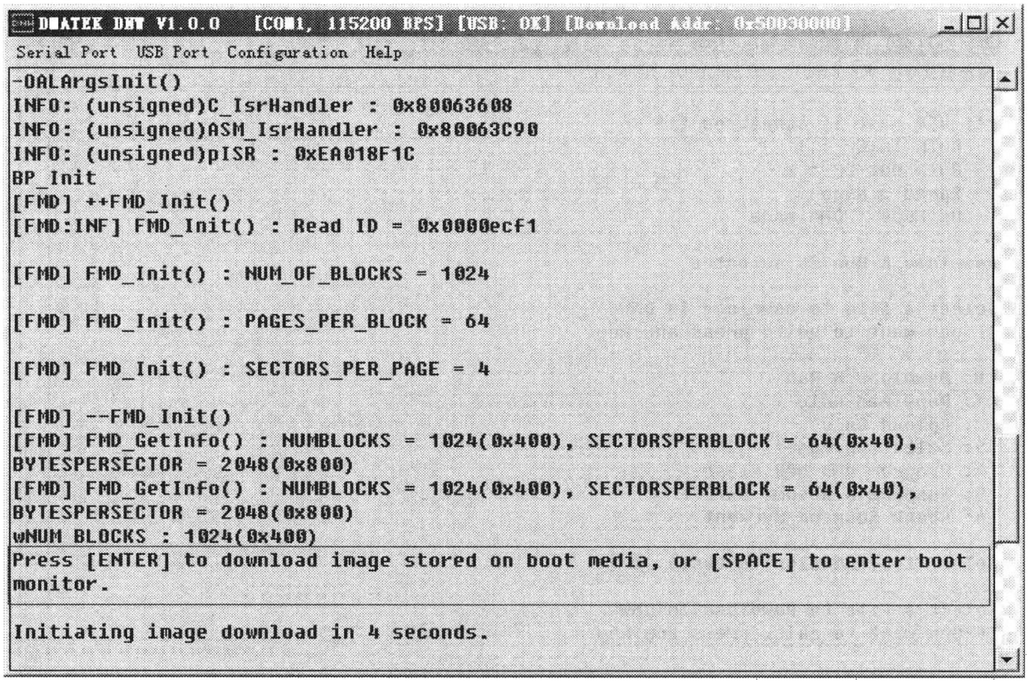



图2-77 下载完毕

11）按下PC主机上的SPACE（空格）键，就会显示“Ethernet Boot Loader Configuration”，如图2-78所示。

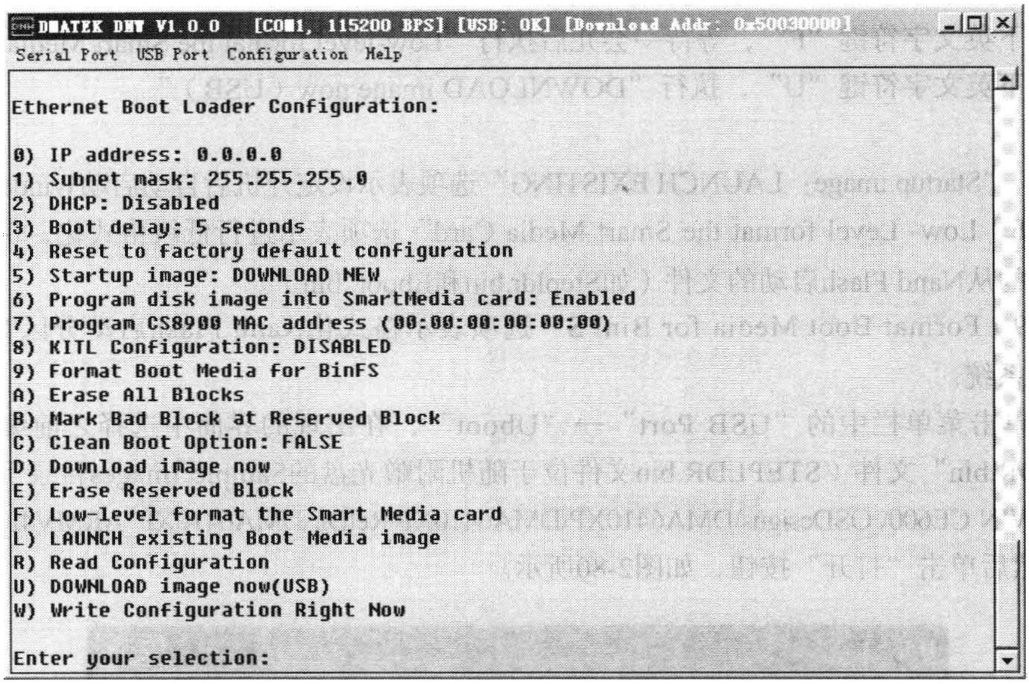



图2-78　配置“Ethernet Boot Loader Configuration”窗口

12）按如下参数配置“Ethernet Boot Loader Configuration”，如图2-79所示。

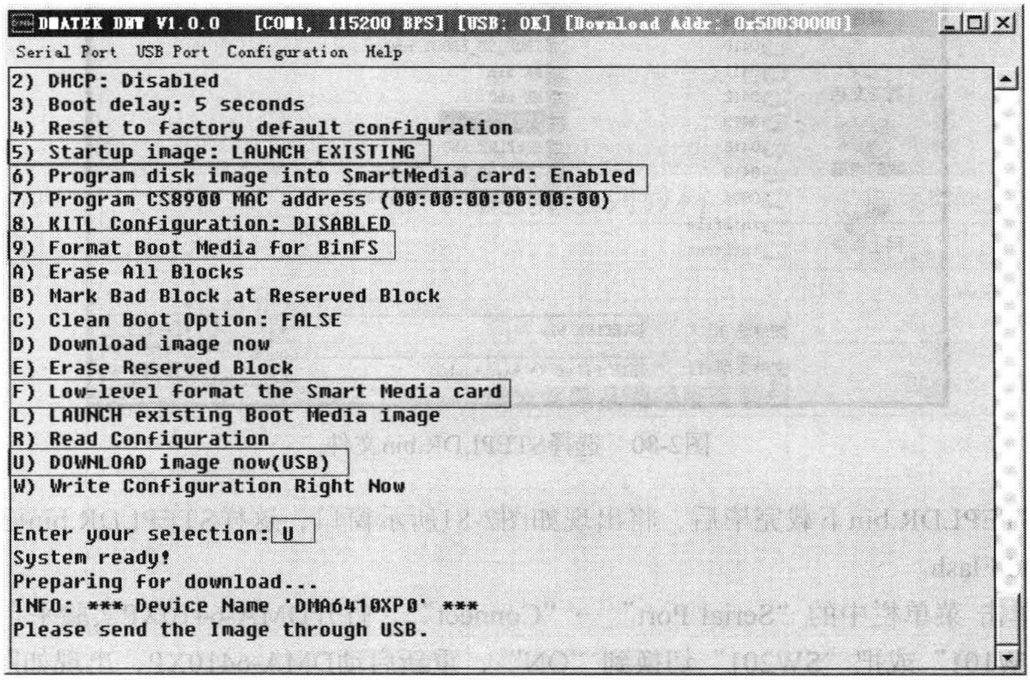



图2-79　配置“Ethernet Boot Loader Configuration”

- 配置“5）Startup image：LAUNCH EXISTING”。
- 配置“6）Program disk image into SmartMedia card：Enabled”。
- 按下数字键“9”，等待一会儿后执行“Format Boot Media for BinFS”。
- 按下英文字符键“F”，等待一会儿后执行“Low-level format the Smart Media card”。
- 按下英文字符键“U”，执行“DOWNLOAD image now（USB）”。

说明：

- 5）“Startup image：LAUNCH EXISTING”选项表示设定开机后自动启动Windows CE。
- “F）Low- Level format the Smart Media Card”选项表示进行低级格式化，保留之前烧写的从Nand Flash启动的文件（如Stepldr.bin和Eboot..bin）。
- “9）Format Boot Media for BinFS”选项表示格式化Nand Flash启动分区为BinFS文件系统。

13）单击菜单栏中的“USB Port”→“Uboot”，在出现的界面中选择之前编译生成的“STEPLDR.bin”文件（STEPLDR.bin文件位于随机附赠光盘的Sample Images目录下或前面编译生成的\WIN CE600\ OSDesigns\DMA6410XP\DMA6410XP\RelDir\DMA6410XP_ARMV4I_Release目录下），然后单击“打开”按钮，如图2-80所示。

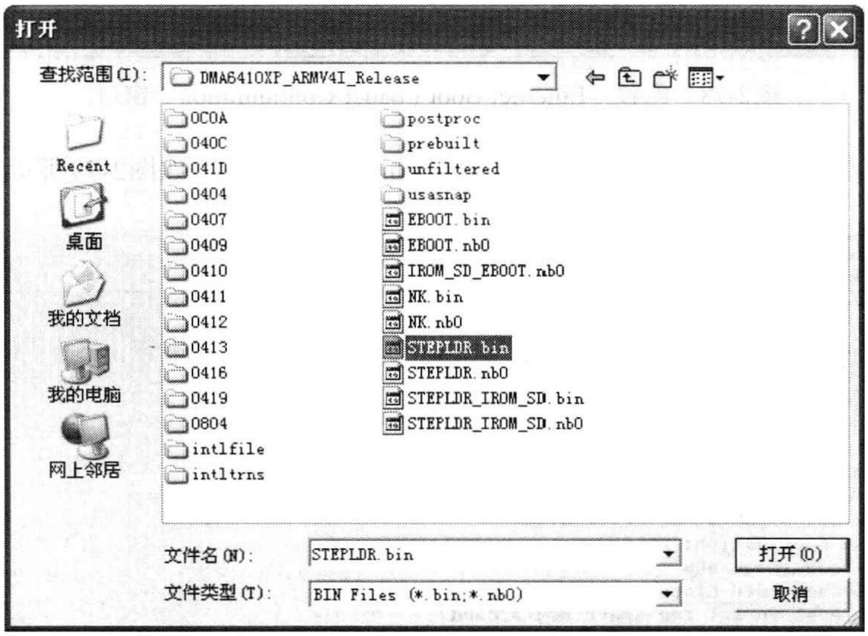



图2-80 选择STEPLDR.bin文件

14）STEPLDR.bin下载完毕后，将出现如图2-81所示窗口，这样STEPLDR.bin已经烧录并存储至Nand Flash。

15）单击 菜单栏中的“Serial Port”→“Connect”，打开DMA-6410XP实验平台的重新开机按钮“SW101”或把“SW201”切换到“ON”，重新启动DMA-6410XP，出现如图2-82所示窗口。

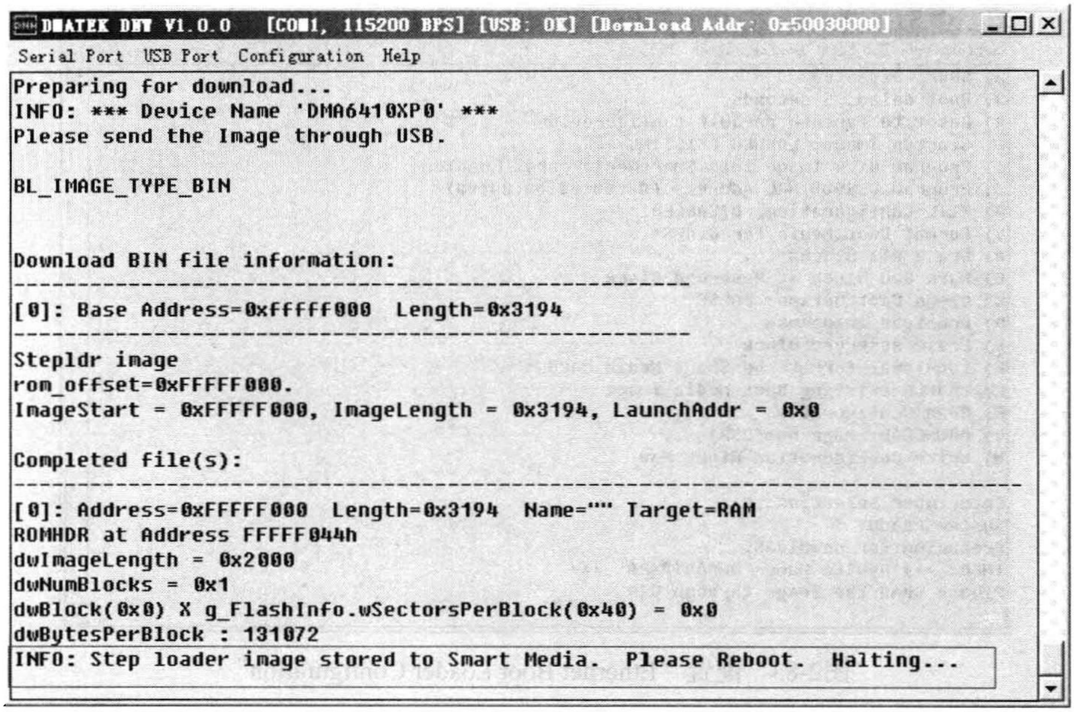



图2-81 下载完毕

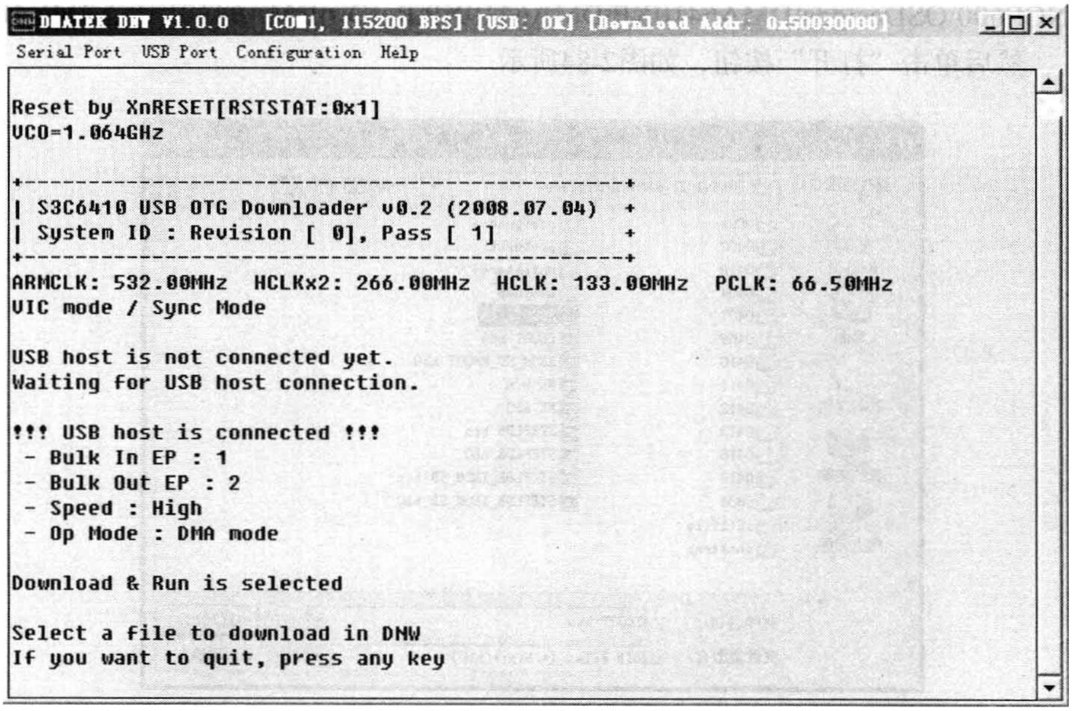



图2-82 重启成功

16）重复前面的步骤7～步骤12，按图2-83所示，配置“Ethernet Boot Loader Configuration”。

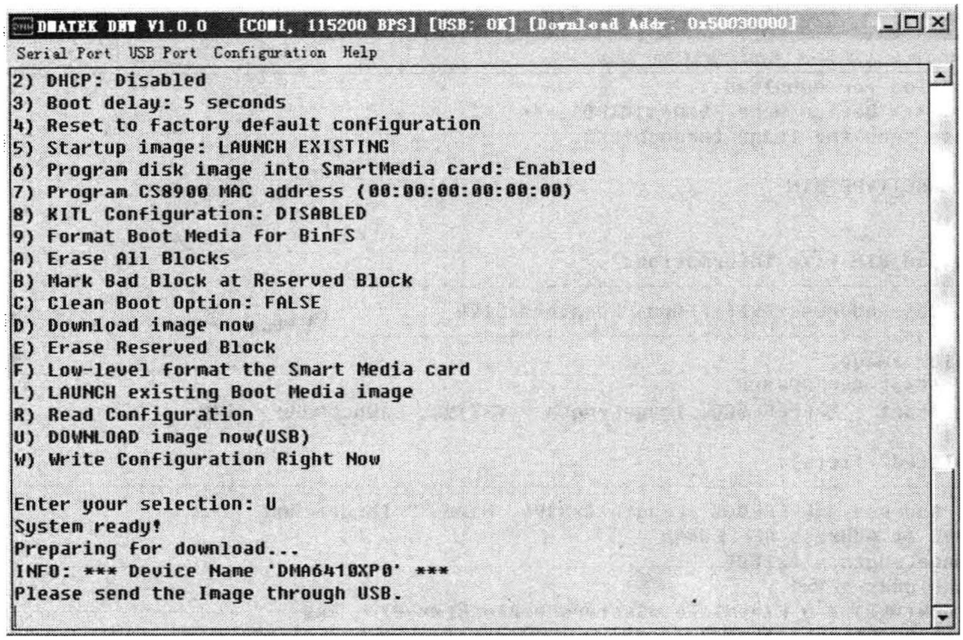



图2-83 配置“Ethernet Boot Loader Configuration”

17）单击菜单栏中的“USB Port”→“UBOOT”，在出现的界面中选择之前编译生成的“EBOOT.bin”文件（EBOOT.bin文件位于随机附赠光盘的Sample Images目录下或前面编译生成的\WINCE600\OSDesigns\DMA6410XP\DMA6410XP\RelDir\DMA6410XP_ARMV4I_Release目录下），然后单击“打开”按钮，如图2-84所示。

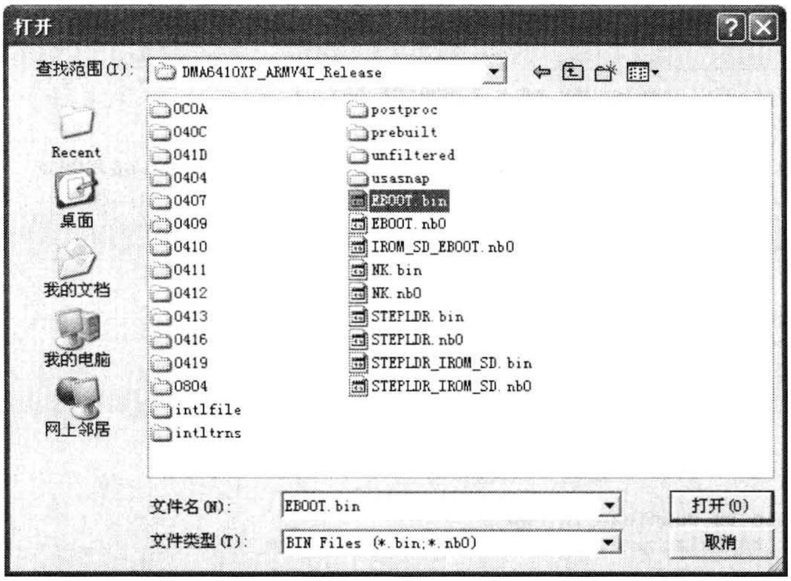



图2-84 选择EBOOT.bin文件

18）EBOOT.bin下载完毕后，将出现如图2-85所示窗口，这样EBOOT.bin已经烧录并存储至Nand Flash。

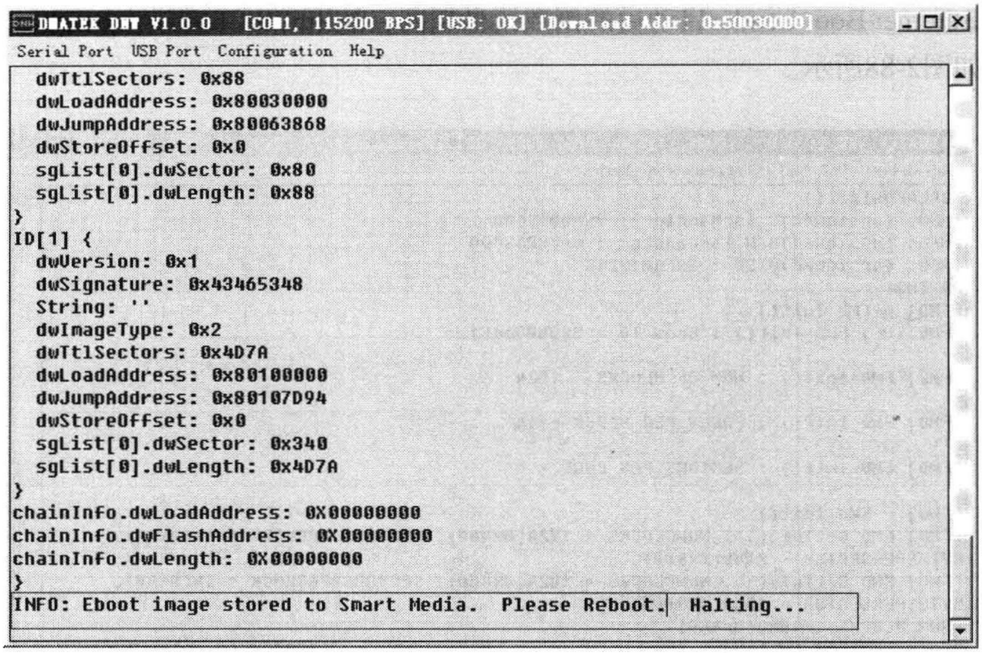



图2-85　下载完毕

19）关闭电源开关SW201，拔掉USB OTG线，将DMA-6410XP实验平台目标板SW118设置成“00100000”，如图2-86所示。选择从Nand Flash启动EBOOT。

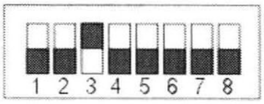



图2-86　设置目标板SW118

20）打开DM-6410XP的电源开关SW201，出现如图2-87所示窗口。

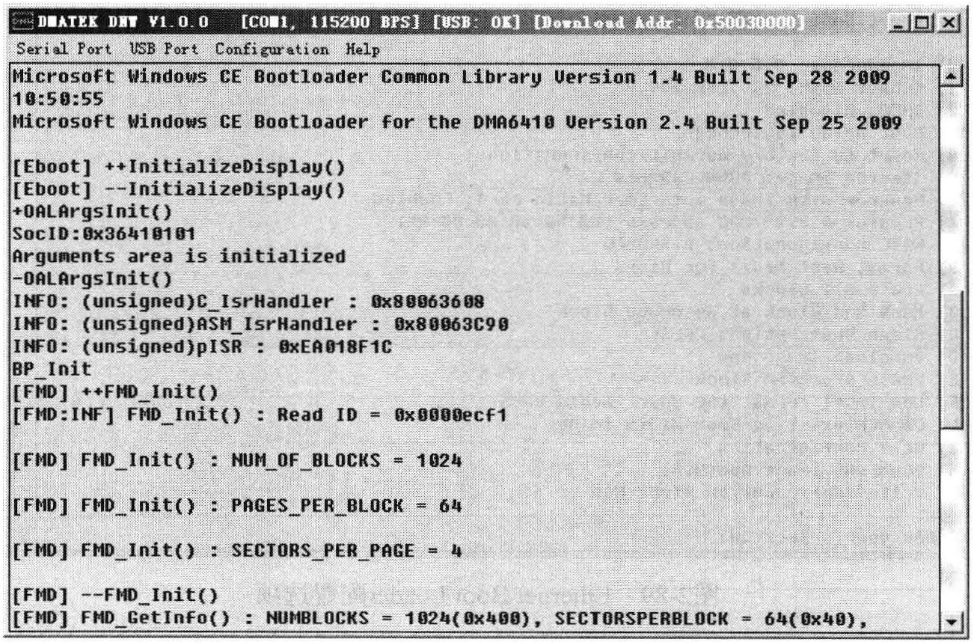



图2-87　从Nand Flash启动EBOOT

21）在Ethernet Boot Loader启动后的倒数读秒期间，按下SPACE键进入Ethernet Boot Loader配置界面，如图2-88所示。

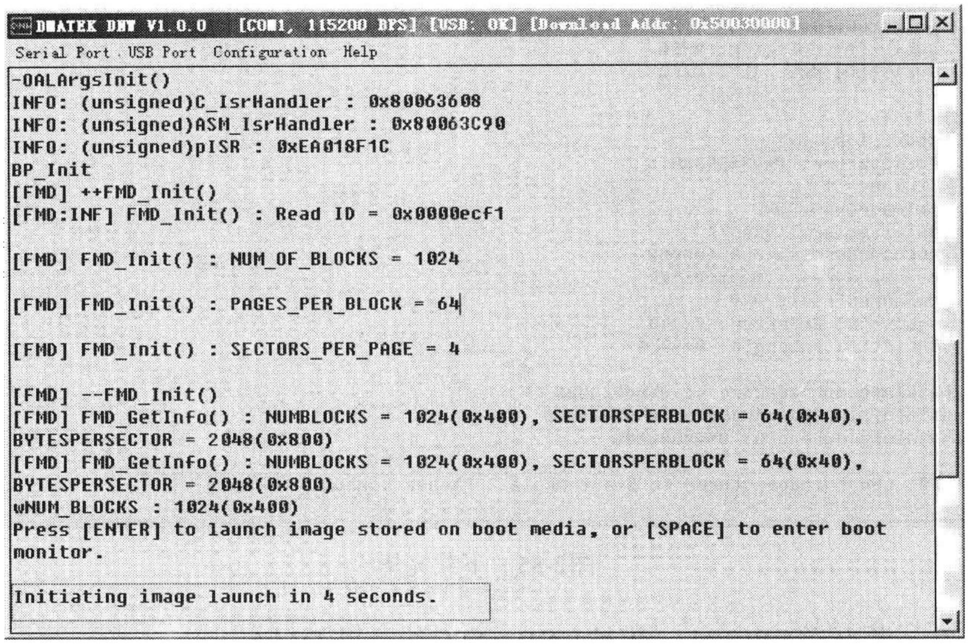



图2-88 进入Ethernet Boot Loader配置界面

22）按如图2-89所示配置Ethernet Boot Loader。

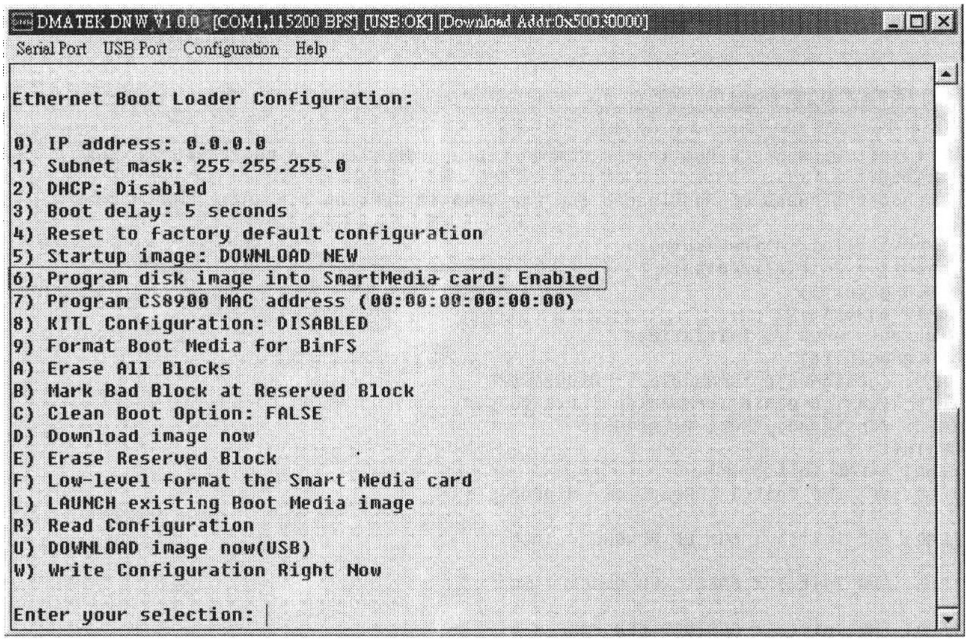



图2-89 Ethernet Boot Loader配置选项

23）在Ethernet Boot Loader配置界面中，按下英文字符键“U”执行“Download image now

（USB）”，然后插入随机附带的USB OTG线，如图2-90所示。

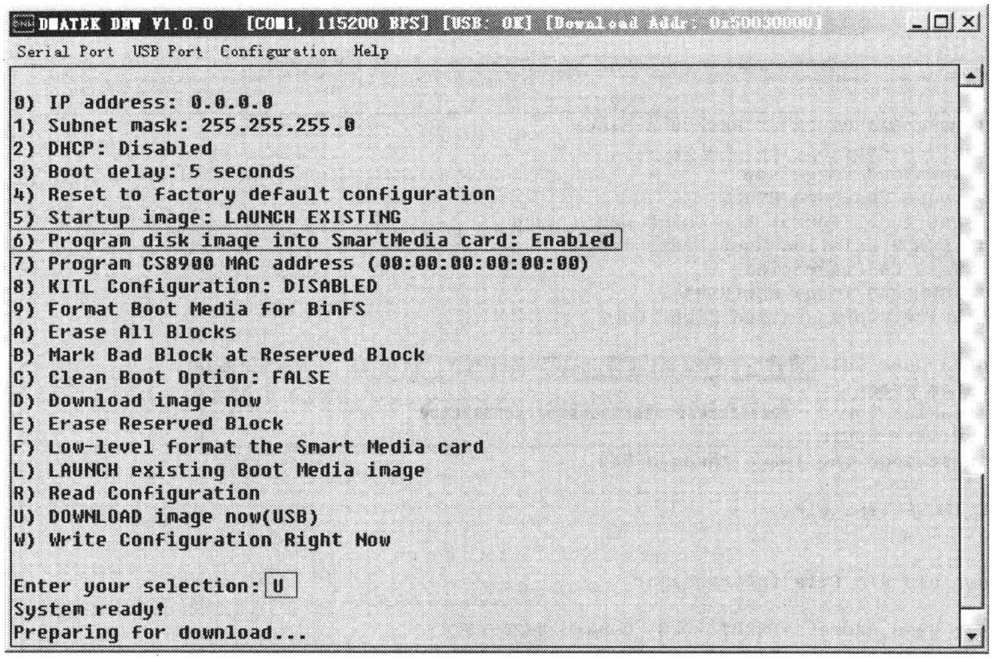



图2-90　执行“DOWNLOAD image now（USB）”

24）单击菜单栏中的“USB Port”→“UBOOT”，在出现的界面中选择之前编译生成的“NK.bin”文件（NK.Bin文件位于随机附赠光盘的Sample Images目录下或前面编译生成的\WINCE600\OSDesigns\DMA6410XP\DMA6410XP\RelDir\DMA6410XP_ARMV4I_Release目录下），然后单击“打开”按钮，如图2-91所示。

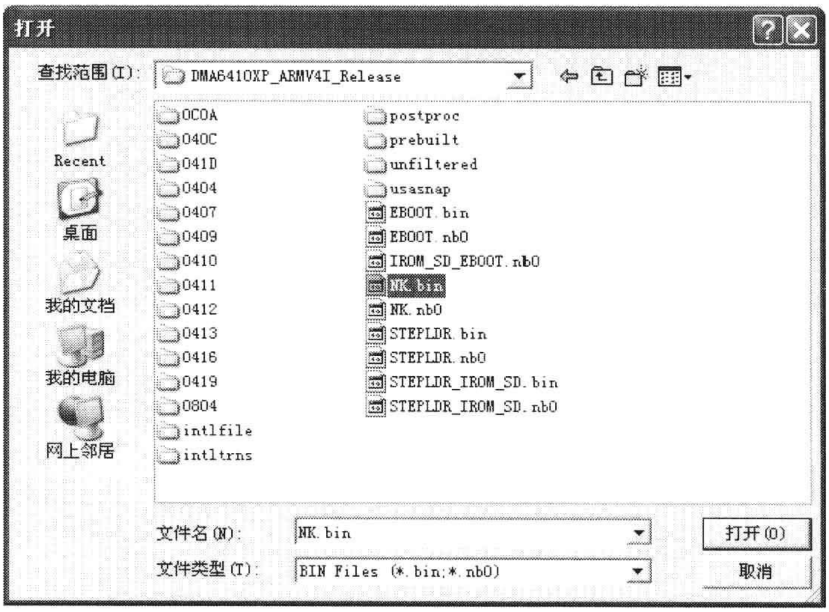



图2-91　选择NK.bin文件

25）大约几分钟NK.bin烧录完成，之后DMA-6410XP将自动执行Windows CE，如图2-92所示。

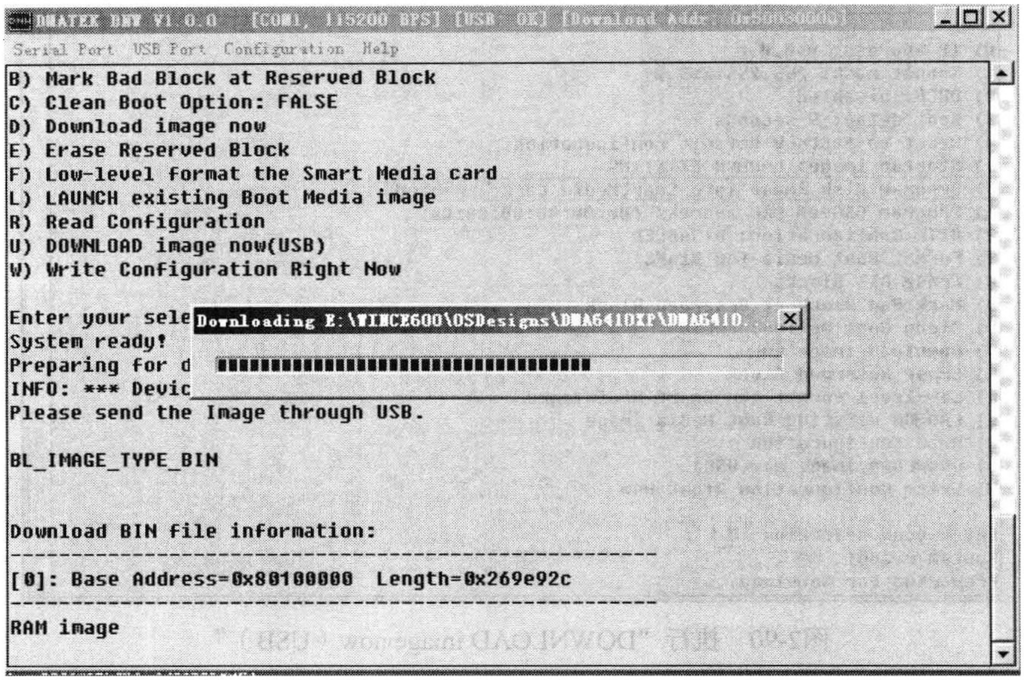



图2-92 自动执行Windows CE

# 第3章　Windows CE应用程序开发

## 3.1　第一个WinCE程序

### 3.1.1　SDK的创建与安装

Visual Studio是一套完整的开发工具集，用于生成ASP.NET Web应用程序、XML Web Service、桌面应用程序等。Visual Basic、Visual C++、Visual C# 和Visual J# 全都使用相同的整合开发环境（IDE）。接下来介绍的所有应用程序都将在Visual Studio 2005 IDE下，使用Visual C++语言进行开发。

SDK（Software Development Kit，软件开发工具包）是软件开发厂商或者组织提供给二次开发人员的特定性开发工具。在Windows Embedded CE 6.0中，操作系统定制厂商为了让计划在该操作系统上进行开发的开发人员能进行相关开发，发布了一个相应的SDK。SDK不仅为该特定的Windows Embedded CE 6.0操作系统开发提供更简易的方式，还简化了开发人员的调试工作。通过安装和使用SDK，即使没有其他Windows Embedded CE 6.0开发工具，例如Platform Builder for CE 6.0和Visual Studio 2005，也可以进行基于该Windows Embedded CE 6.0系统的应用程序开发。例如，微软公司发行了Windows Mobile和Windows Automotive，专门为二次开发人员提供了SDK，当然，这两个SDK的意义要更广泛。

一般情况下，一个完整的SDK包含以下几方面内容：

- 头文件Header
- 库文件Library
- 连接支持文件Connectivity Files
- 平台管理器Platform Manager
- 执行时库文件Run-Time Files
- 文件Documentation

首先要明白SDK究竟包含一些什么东西？在这里重点要说明的是SDK中所包含的头文件和库文件，这些文件都是当前定制系统所包含的文件，并不是整个Windows Embedded CE 6.0中所包含的文件，它是整个Windows Embedded CE 6.0的一个子集。这是因为制作的系统就是Windows Embedded CE 6.0的一个子集。可以使用图3-1来表示它们之间的关系。

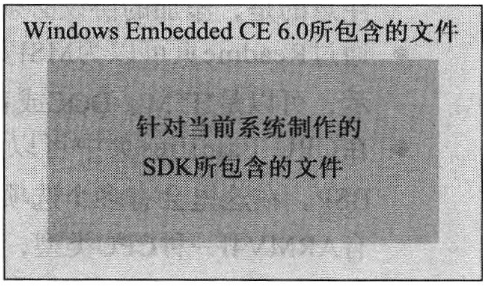



图3-1　SDK文件包含关系

其次是为什么要制作一个SDK？这完全是为了在当前系统上更加便利地开发应用程序。在开发应用程序的时候会调用系统的一些头文件和库文件等系统资源。这些资源都包含在相应的SDK中。如果能找到一个全面的SDK，该SDK包含了Windows Embedded CE 6.0中的所有文件，它并不是子集。那么完全可以在该SDK的基础上开发应用程序，但是这时开发的应用程序会出现一个问题：如果应用程序调用了当前系统以外的一些头文件、库文件或其他资源，那么当把该程序放到当前系统上时，将有可能因为缺少资源而无法执行。所以为了避免出现这种情况，需要针对当前的系统开发一个SDK，然后在该SDK的基础上开发应用程序，这样开发出来的应用程序就可以避免发生上述情况。

使用Platform Builder for CE 6.0提供的SDK创建工具可以创建一个针对性的SDK，但是，前提是已经创建了一个可以成功执行的基于Windows Embedded CE 6.0的操作系统。首先打开以前制作的Windows Embedded CE 6.0的项目。

在Visual Studio 2005中依次单击“Project”→“Add New SDK”，打开SDK属性页，如图3-2所示，然后就可以开始对一个SDK进行配置了。

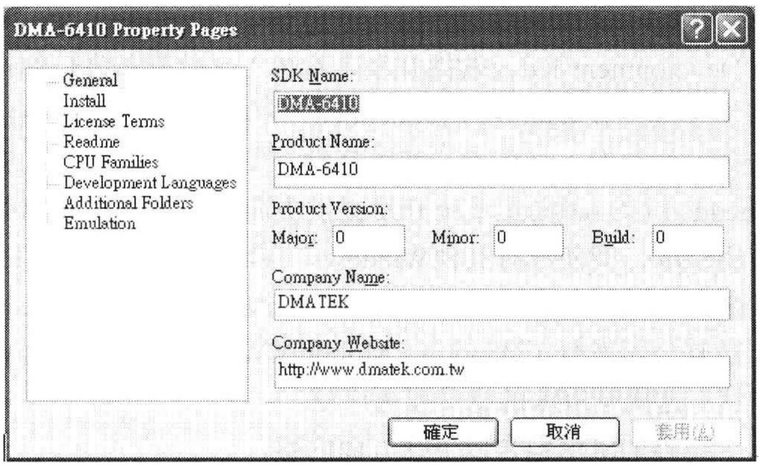



图3-2 SDK属性页

通过该SDK属性页，可以自定义SDK的大部分特性。下面介绍一下各个属性。

- Install页用来配置生成的MSI安装程序的相关信息，如文件名和语言。
- License Terms页包括最终用户许可协议（End-User License Agreement，EULA）。但要注意的是，添加的协议必须事先使用WordPad等文件编辑工具编辑成RTF文件。
- 通过Readme页可以为MSI安装程序添加一个Readme文件，这个文件会在安装完成后显示，可以是HTM、DOC或者RTF文件。
- 在CPU Families页中可以配置使用的CPU类型，如果在创建项目的时候选择了多个BSP，在这里会有多个选项，开发人员可以对SDK支持的CPU类型进行选择，在这里只有ARMV4I一种CPU类型，并且是被选中的。
- 在Development Languages页中将开发语言分为两类，本地开发语言和托管开发语言。这两种语言可以二选一，也可以两个全选。

- 在Emulation页中可以对仿真器进行配置，如果在创建这个基于Windows Embedded CE 6.0的嵌入式操作系统时选择了Emulation支持，那么在Emulation中可以对仿真器进行选择，还可以对其进行一定的配置，如配置仿真器的内存、显示属性等。不使用仿真器支持也是可行的。

可以添加多个SDK。添加的所有SDK会显示在解决方案管理器的SDK目录树中。右击某个SDK，可以在弹出的快捷菜单中对其进行再配置、删除和生成等操作，如图3-3所示。也可以在Visual Studio 2005中依次单击"Build"→"Build All SDKs"菜单项生成所有的SDK。SDK在生成时可以通过Visual Studio 2005的输出窗口观察其生成过程与状态。

图3-3 在解决方案管理器中配置SDK

SDK的生成过程会持续几分钟，生成的*.msi文件存放在SDK属性页中Install页的"MSI Folder Path"所指向的文件夹。这样就完成了SDK的配置，接下来可以在该SDK基础上开发应用程序了。

在安装SDK的时候，如果出现了一个提示安装"Visual Studio 2005 Service Pack 1"的对话框，那么应先安装该软件，否则在安装好SDK后使用该SDK编译应用程序时会出错。

### 3.1.2 应用程序设计

本小节介绍如何设计应用程序，具体步骤如下。

1）启动Visual Studio 2005，依次单击"File"→"New"→"Project"，打开新建项目对话框。如图3-4所示，可以设置该项目的名字和存放的路径，在"Project types"中选择"Visual C++"目录下的"Smart Device"，然后在"Templates"中选择"MFC Smart Device Application"，制作一个在智能设备上运行的MFC应用程序。确定好项目名字和路径之后单击"OK"按钮。

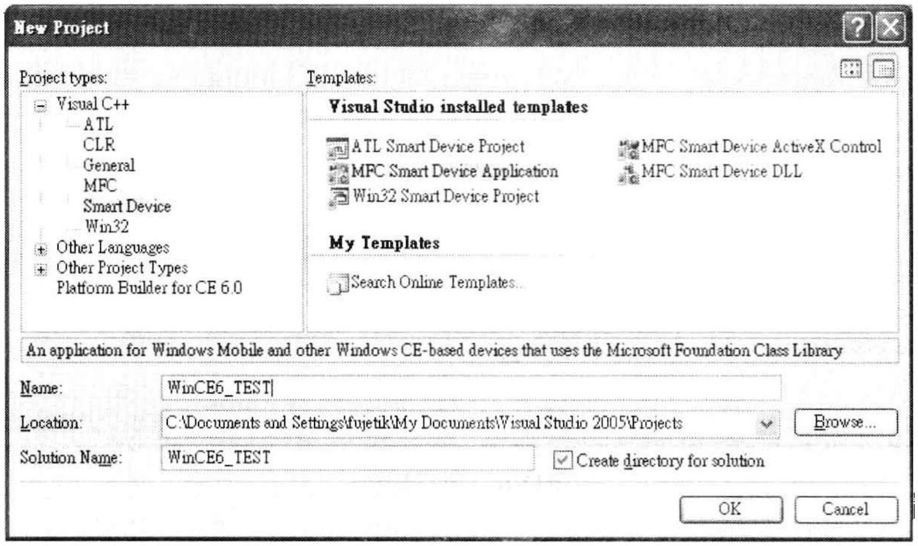


图3-4 新建智能设备上运行的MFC应用程序

2）单击“OK”按钮后再单击“Next”按钮进入SDK选择界面，如图3-5所示。系统预设添加了“Pocket PC 2003”这个SDK，在这里并不需要它，移除它，把刚才制作的“DMA-6410”添加过来。然后单击“Next”按钮。

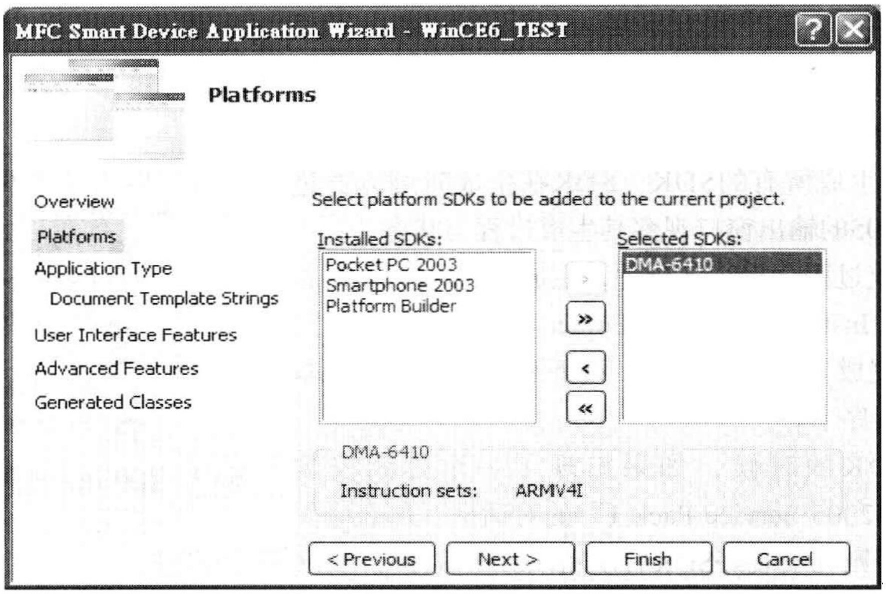



图3-5 SDK选择

3）单击“Next”按钮进入项目配置页面，如图3-6所示。单击“Dialog based”单选项，使用静态链接库“Use MFC in a static library”，语言选择“中文（中国）”。此后的配置可以使用默认选项，在此可以单击“Finish”按钮完成应用程序新建工程。

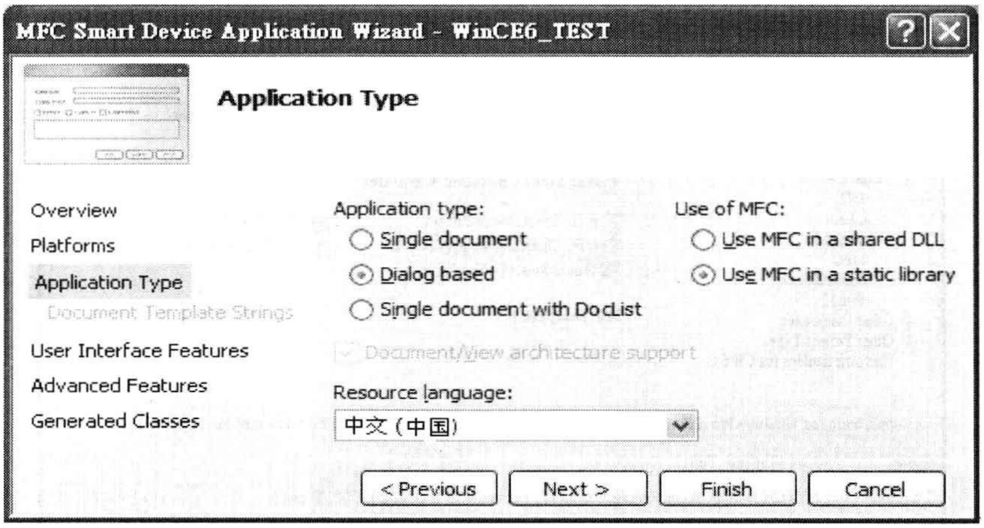



图3-6 项目配置

4）在左侧“解决方案管理器”中单击可以查看所有文件，如图3-7所示。在所有文件列

表中双击“Win_CE6_Test.rc”，打开资源视图。

5）在资源视图模式下单击“Dialog”→“IDD_Win_CE6_TEST_ DIALOG”，可以看到一个很熟悉的窗口。这和VC中的MFC开发类似，如图3-8所示。

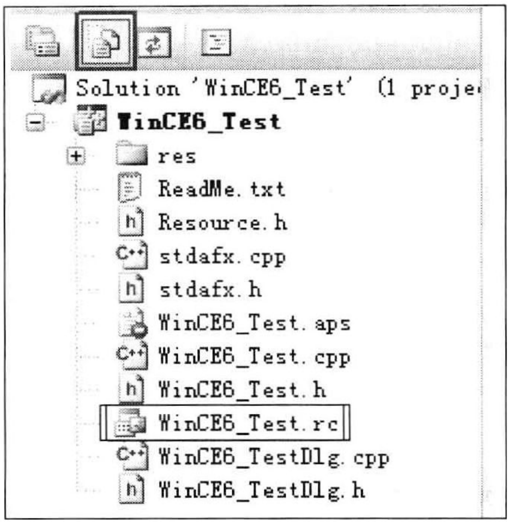



图3-7 解决方案管理器

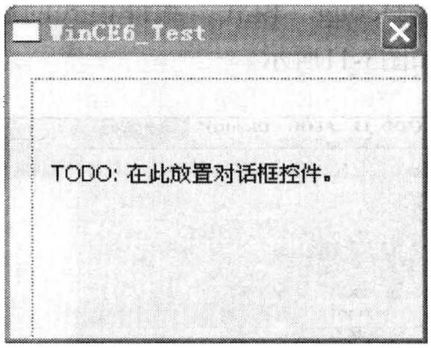



图3-8 WINCE6_Test应用程序

6）依次单击“View”→“ToolBox”菜单项，将出现一个“控件”窗口，这里有很多控件，用鼠标单击其中的“Button”控件，如图3-9所示。

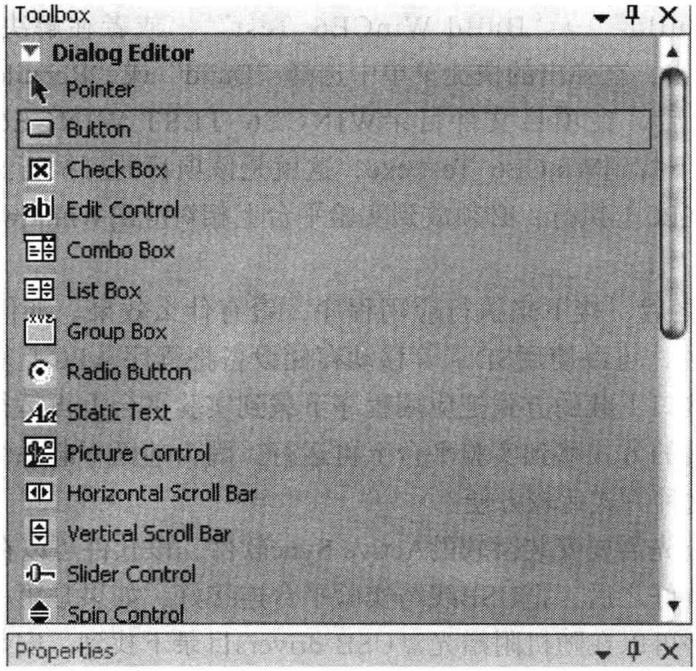



图3-9 按钮控件

7）然后单击对话框的窗体，将出现一个按钮，可以移动和改变该按钮的位置和大小，如

图3-10所示。

8）在该按钮上右击，在弹出的快捷菜单中选择“Properties”，将出现该按钮的属性窗口，在这里可以设置该按钮的一些属性。设置“ID”属性为“IDC_BUTTON_HelloCE”，“Caption”属性为“HelloCE”。再添加一个按钮，设置其“ID”属性为“IDC_BUTTON_Shut Down”，“Caption”属性为“Shut Down”。

在这里保持预设的程序结构，不对程序做任何修改。可以修改编译的方式为“Release”，这样可以使生成的可执行文档更小，但也会缺少一些除错信息，如果是在程序的除错阶段可以使用“Debug”模式。编译模式的后面显示的是当前的SDK，该SDK正是前面制作的那个SDK，如图3-11所示。

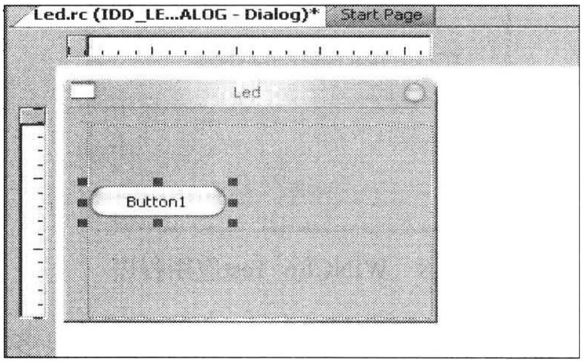



图3-10 按钮控件的布局

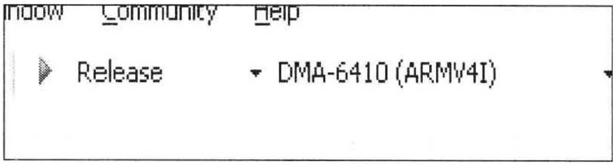



图3-11 编译模式与SDK选择

9）依次单击“Build”→“Build WinCE6_Test”，或者在解决方案管理器中项目名“WinCE6_Test”上右击，在弹出的快捷菜单中选择“Build”或“Rebuild”。

10）编译完成之后，在项目文件目录WINCE6_TEST\WINCE6_TEST\ DMA-6410（ARMV4I）\Release下找到WinCE6_Test.exe，这就是该项目的可执行文件，但要注意的是，该可执行文件并不能在PC上执行，必须放到实验平台上相对应的Windows CE 6.0系统中才能正确执行。

制作好应用程序之后，接下来执行应用程序，看有什么效果。如何把应用程序放到实验平台上呢？方法有多种，可以使用SD卡等移动存储设备把程序从PC上复制到实验平台上再运行，也可以使用网络接口下载的方式把应用程序下载到实验平台上再运行，在这里介绍第三种方法，即通过USB下载的方式移到实验平台上再运行。因为前面下载系统时也是通过USB实现的，所以在这里使用这种方式比较方便。

使用这种方法，首先需要安装微软的Active Sync软件，该软件可以在随机附赠的光盘Tools目录下找到。软件安装好之后，把USB线与实验平台连接好，如果是第一次使用，会提示安装驱动程序，该驱动程序可以在随机附赠光盘USB drivers目录下找到。安装好驱动程序后会自动在实验平台和桌面之间建立一个连接，如果没有正确连接，请重新插一下USB连接线。

系统连接好之后，在“我的电脑”中会多出一个“其他设备”，操作该设备就如同操作其他盘符一样，这样就可以把应用程序通过复制的方式放到实验平台上。在实验平台上双击执行

应用程序，看看制作的第一个应用程序是什么样的。

### 3.1.3 使用C#编写LED驱动程序

本小节介绍使用C#语言来编写一个LED的驱动程序。

#### 1. 创建一个智能设备上的C#项目

启动Visual Studio 2005，依次单击“File”→“New”→“Project”，将打开新建项目界面。在该界面左侧“Project types”栏中依次单击“Other Languages”→“Visual C#”→“Smart Device”→“Windows CE 5.0”，在右侧“Templates”栏中单击“Device Application”，如图3-12所示，最后在下面指定要创建的项目的名称LED_CSharp和存放的路径，全部设置好后单击“OK”按钮即可完成项目的创建。如果在“Other Languages”中不能找到“Visual C#”的选项，请先确认在安装Visual Studio 2005的时候是否安装了有关C#的组件等。

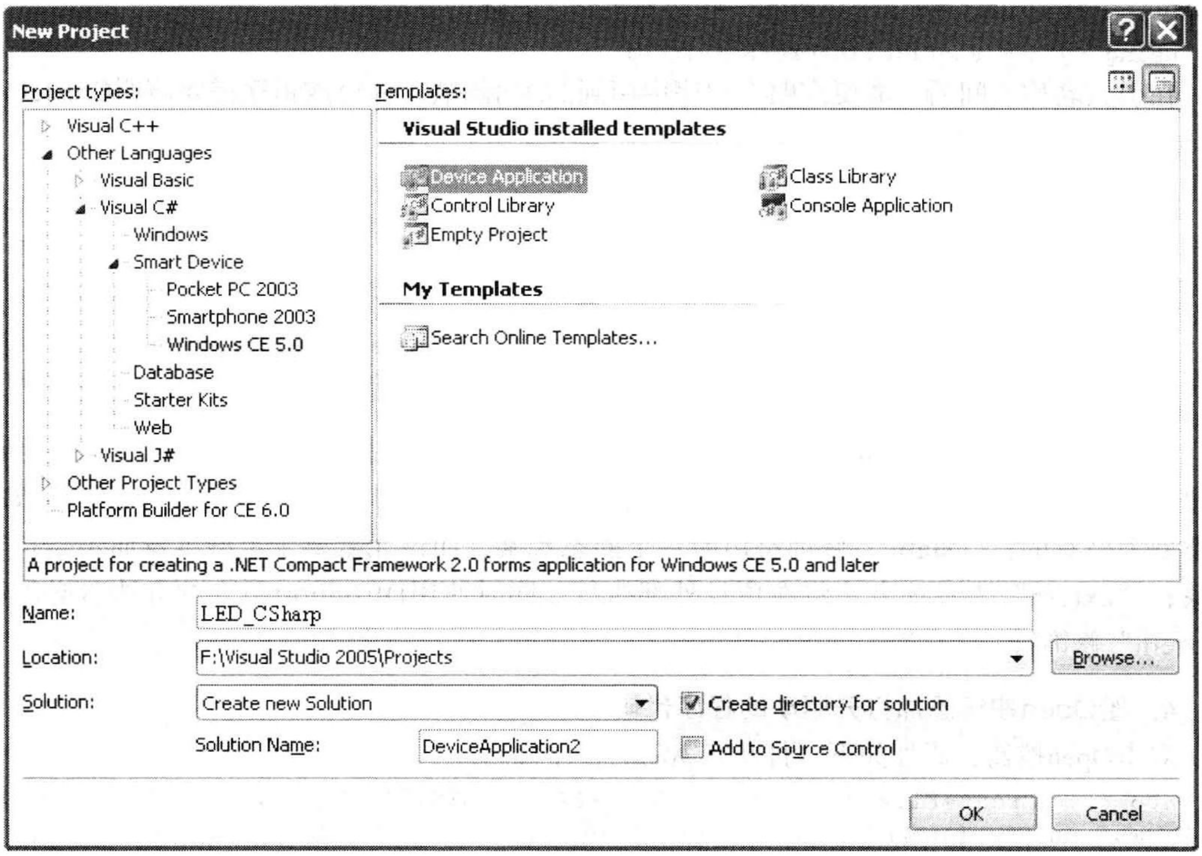



图3-12 新建C#项目

#### 2. 制作软件接口

在窗体界面中，先调整窗体的大小，然后再加上三个按钮，分别命名为“Open”、“ON”和“OFF”。制作类似图3-13所示的界面。

Open按钮用于打开驱动，ON和OFF按钮用于开关LED灯。

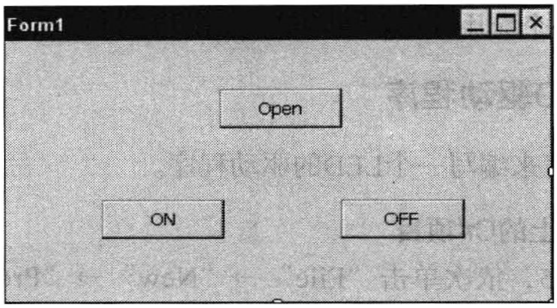


图3-13 窗体的布局

### 3. 声明API函数

由于该软件中要使用到系统API来打开驱动和控制LED灯的亮灭，所以需要先声明这些API。

首先，在调用API之前，必须先引入System.Runtime.InteropServices这个名称空间。该名称空间包含了在Visual C#中调用API的一些必要集合，具体的方法如下：

```
using System.Runtime.InteropServices;
```

在引入名称空间后，需要声明在程序中用到的API函数。针对该程序需要声明CreateFile和DeviceIoControl两个API。

```
[DllImport("coredll.dll")]
public static extern IntPtr CreateFile(String lpFileName,UInt32 dwDesiredAccess,UInt32
dwShareMode,IntPtr lpSecurityAttributes,UInt32 dwCreationDisposition,UInt32
dwFlagsAndAttributes,IntPtr hTemplateFile);
[DllImport("coredll.dll")]
public static extern bool DeviceIoControl(IntPtr hDevice,UInt32 dwIoControlCode,Byte[]
lpInBuffer,UInt32 nInBufferSize,Byte[] lpOutBuffer,UInt32 nOutBufferSize,UInt32
lpBytesReturned,IntPtr lpOverlapped);
```

在上述代码中，“DllImport”属性用来从不可控程序代码中调用一个方法，它指定了DLL的位置，该DLL中包含调用的外部方法；“coredll”设定了类库名；“public”指明函数的访问类型为公有的；“static”修饰符声明一个静态元素，而该元素属于类型本身而不是指定的对象；“extern”表示该方法将在项目外部执行，同时使用DllImport引入的方法时必须使用“extern”修饰符。

### 4. 给Open按钮添加打开驱动的程序代码

双击Open按钮，添加如下“打开”驱动的程序代码：

```
hPort = CreateFile("LED1:",GENERIC_READ | GENERIC_WRITE,0,IntPtr.Zero,OPEN_
EXISTING,0,IntPtr.Zero);
if(hPort == (IntPtr)INVALID_HANDLE_VALUE)
{
    MessageBox.Show("Open Led Driver Fail");
}
```

### 5. 给ON按钮添加点亮LED灯的程序代码

双击ON按钮，添加如下点亮LED灯的程序代码：

```
byte[] led1 = new Byte[1];
```

```
edl[0] = 0;
byte[] pout = new Byte[10];
DeviceIoControl(hPort,LED_ON,led1,0,pout,10,0,IntPtr.Zero);
```

#### 6. 给OFF按钮添加关闭LED灯的程序代码

双击OFF按钮，添加如下熄灭LED灯的程序代码：

```
byte[] led1 = new Byte[1];
led1[0] = 0;
byte[] pout = new Byte[10];
DeviceIoControl(hPort,LED_OFF,led1,0,pout,10,0,IntPtr.Zero);
```

#### 7. 编译测试该软件

编译该软件之后，把可执行文件复制到实验平台上，双击执行它，先单击Open按钮，然后分别单击ON按钮和OFF按钮，看能否控制LED灯的亮灭。

### 3.1.4 使用MFC编写界面程序

#### 1. 创建MFC框架应用程序项目

首先启动Visual Studio 2005，在菜单栏依次单击“File”→“New”→“Project”，出现“New Project”界面，在该界面左侧选择“Visual C++”→“Smart Device”。在右侧选择“MFC Smart Device Application”。在下侧的“Name”栏填写项目的名称为“TestBitmap”，在“Location”栏选择项目的存放路径，如图3-14所示。

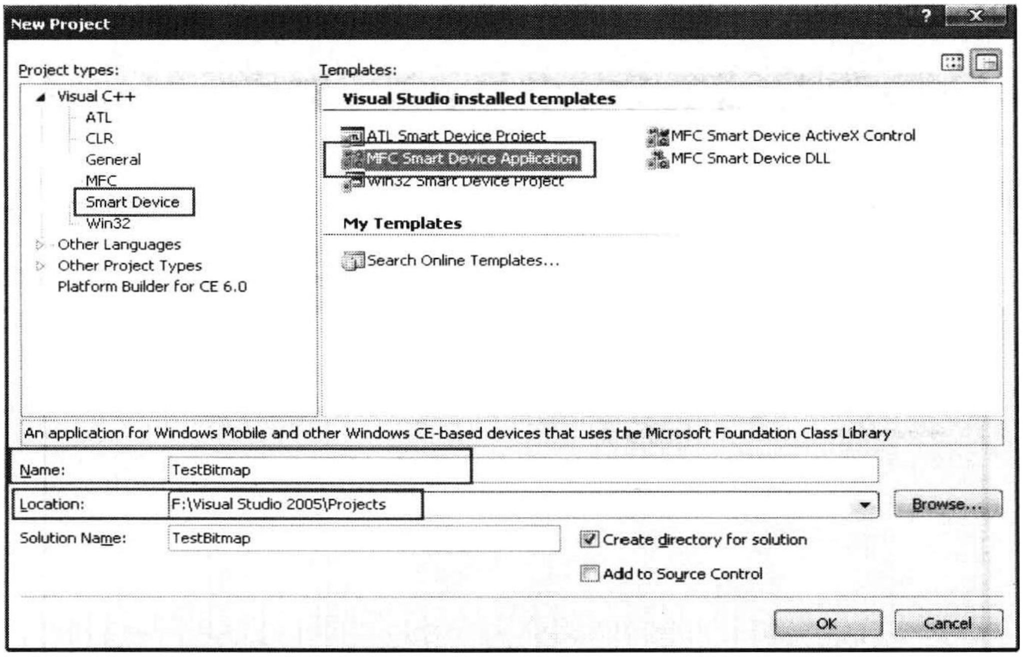



图3-14 创建基于MFC智能设备的应用程序

单击“OK”按钮后单击“Next”按钮进入选择SDK页面。先把项目自动添加的SDK移除，然后添加名为“STANDARDSDK_500”的SDK，如图3-15所示。如果项目创建到这里没有

看到这个SDK，那么请先安装该SDK，该SDK是微软提供的Windows CE 5.0的标准SDK，可以通过搜索在网络上轻松下载该SDK。如果有其他Windows CE系统上的SDK，使用它们也可以。

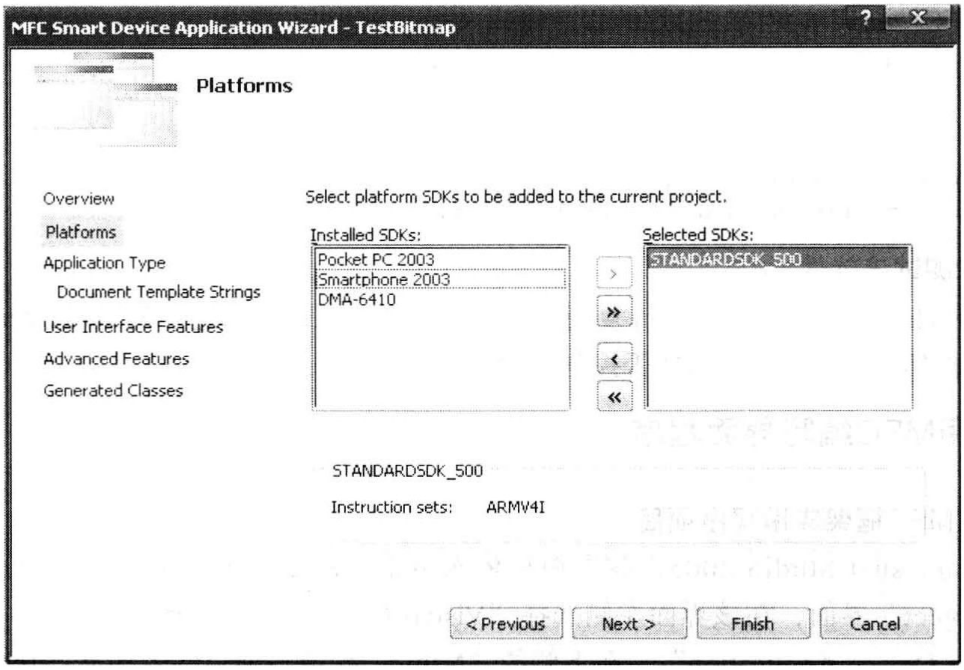



图3-15 选择SDK

单击"Next"按钮进入下一界面，在该界面选择"Dialog based"选项，如图3-16所示。

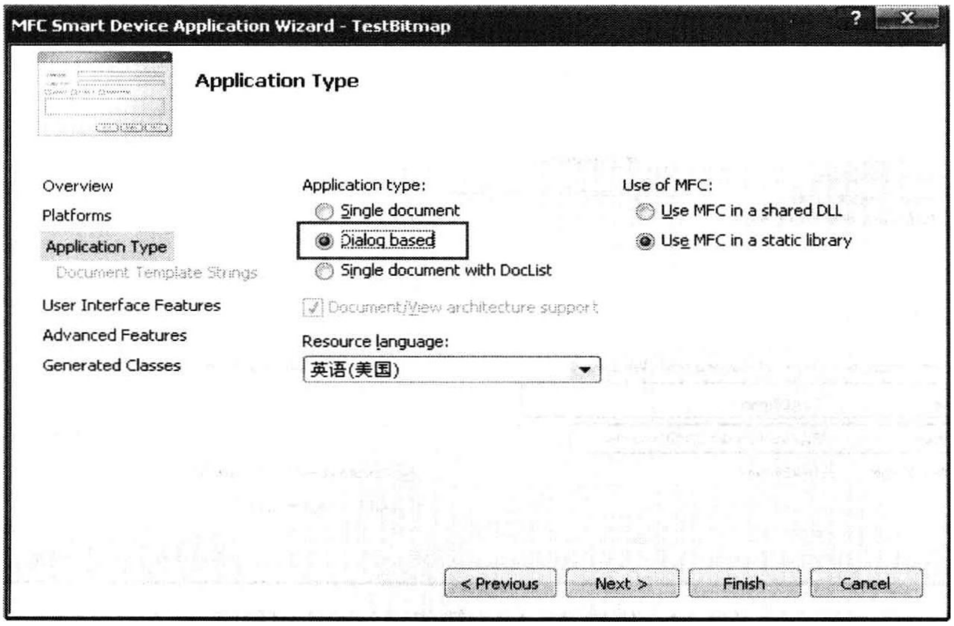



图3-16 选择应用程序的类型

至此单击"Finish"按钮结束创建项目向导，完成创建项目。

编译该项目，在Windows CE系统上运行该程序，将出现类似图3-17所示的界面。

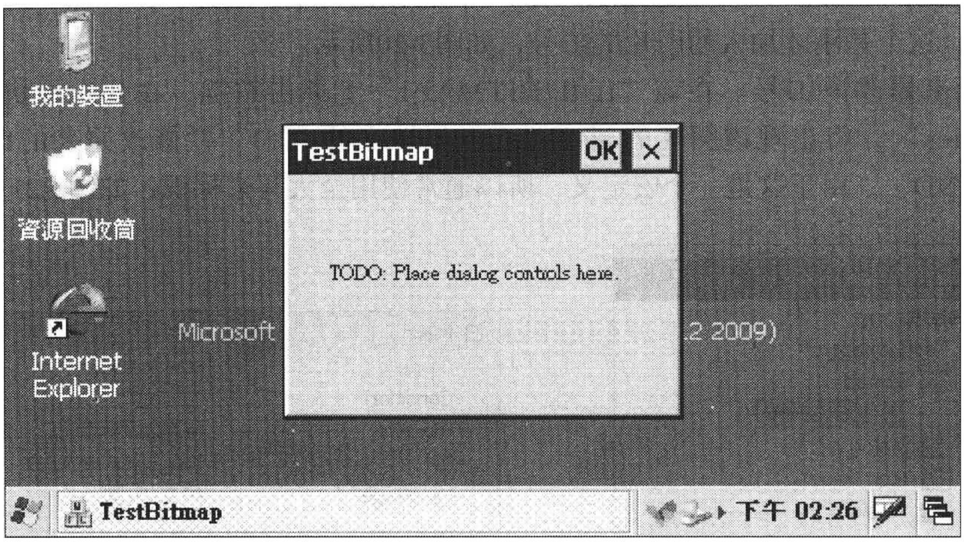



图3-17 程序运行效果

## 2. 添加图片

可以看出，在默认的情况下，创建的程序的背景是灰色的，下面介绍如何将程序背景更改为图片。

1）找一张200×200像素的图片。把该图片放到项目目录下的res文件夹下。

2）在Visual Studio 2005中单击菜单“View”→“Solution Explorer”，在出现的窗口中，单击窗口中的“TestBitmap”，如图3-18所示。

3）单击菜单“Project”→“Add Resources”，出现添加资源对话框。在左侧选择“Bitmap”，在右侧单击“Import”按钮，如图3-19所示。

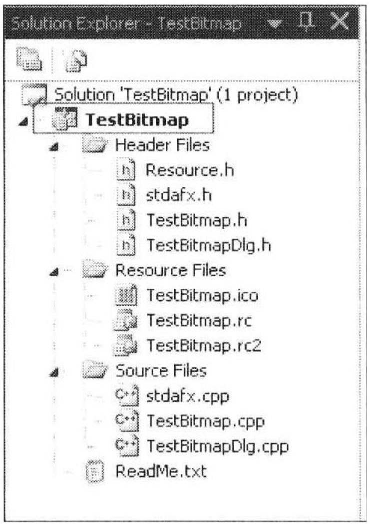



图3-18 解决方案管理器

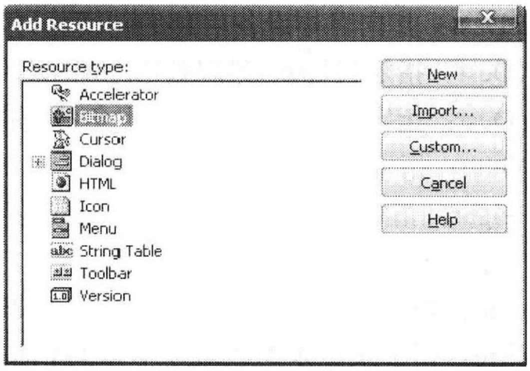



图3-19 添加位图资源

4）单击“Import”按钮后将出现选择文件对话框，选择res文件夹下刚才放入的图片。

5）单击菜单“View”→“Resource View”，在该窗口中，展开“Bitmap”将看到“IDB_BITMAP1”，这个是刚才加入的图片的ID号，如图3-20所示。

6）下面来修改该ID号。在该“IDB_BITMAP1”上单击右键，在弹出的快捷菜单中选择“Properties”，将出现该图片的属性框，在属性框的“ID”栏修改原来的ID为“IDB_BACKGROUND”，由于这是一个宏定义，所以通常使用全大写来标识，如图3-21所示。

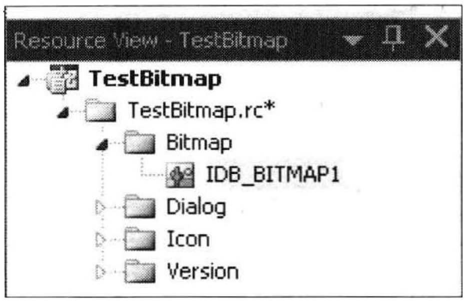


图3-20 查看添加的位图资源

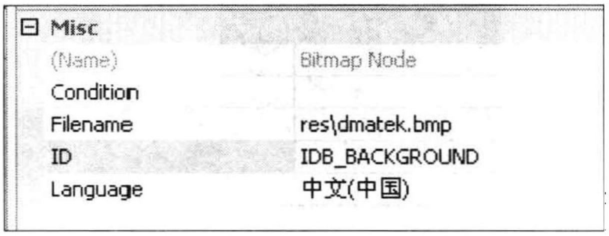


图3-21 修改资源标识符

至此，将一张图片映射到了程序的项目中，接下来则是把这个背景图片显示出来。

### 3. 显示图片

在显示图片之前，需要明白两件事情：如何把位图在程序中显示出来以及这些显示图片的程序代码应该放在哪里。

第一，如何把位图显示出来。

这里显示位图的方式是使用Windows的GDI画图。使用这种方式就需要先来看一个很重要的API函数BitBlt。

**函数功能**：该函数对指定的来源设备信息区域中的像素进行位块（bit_block）转换，以传送到目标设备信息。

**函数原型**：`BOOL BitBlt(HDC hdcDest,int nXDest,int nYDest,int nWidth,int nHeight,HDC hdcSrc,int nXSrc,int nYSrc,DWORD dwRop);`

参数：

- hdcDest：指向目标设备信息的句柄。
- nXDest：指定目标矩形区域左上角的X轴逻辑坐标。
- nYDest：指定目标矩形区域左上角的Y轴逻辑坐标。
- nWidth：指定来源和目标矩形区域的逻辑宽度。
- nHeight：指定来源和目标矩形区域的逻辑高度。
- hdcSrc：指向来源设备信息的句柄。
- nXSrc：指定来源矩形区域左上角的X轴逻辑坐标。
- nYSrc：指定来源矩形区域左上角的Y轴逻辑坐标。

- dwRop：指定光栅操作（raster-operation）的方式。这些方式将定义来源矩形区域的颜色数据如何与目标矩形区域的颜色数据组合以完成最后的颜色。

参数dwRop的一些取值如下所示：

- BLACKNESS：表示使用与物理调色板的索引0相关的色彩来填充目标矩形区域（对预设的物理调色板而言，该颜色为黑色）。
- DSTINVERT：表示使目标矩形区域颜色取反相（invert）。
- MERGECOPY：表示使用布尔型的AND（与）将来源矩形区域的颜色与特定模式组合在一起。
- MERGEPAINT：通过使用布尔型的OR（或）将反向的来源矩形区域的颜色与目标矩形区域的颜色合并。
- NOTSRCCOPY：将来源矩形区域颜色取反相，并复制到目标矩形区域。
- NOTSRCERASE：使用布尔型的OR（或）组合来源和目标矩形区域的颜色值，然后将合成的颜色值取反相。
- PATCOPY：将特定的模式复制到目标位图上。
- PATPAINT：通过使用布尔OR（或）将来源矩形区域取反相后的颜色值与特定模式的颜色合并。然后使用OR（或）将该操作的结果与目标矩形区域内的颜色合并。
- PATINVERT：通过使用XOR（异或）将来源和目标矩形区域内的颜色合并。
- SRCAND：通过使用AND（与）将来源和目标矩形区域内的颜色合并。
- SRCCOPY：将来源矩形区域直接复制到目标矩形区域。
- SRCERASE：通过使用AND（与）将目标矩形区域颜色取反后与来源矩形区域的颜色值合并。
- SRCINVERT：通过使用布尔型的XOR（异或）将来源和目标矩形区域的颜色合并。
- SRCPAINT：通过使用布尔型的OR（或）将来源和目标矩形区域的颜色合并。
- WHITENESS：使用与物理调色板中索引1有关的颜色填充目标矩形区域（对于预设物理调色板来说，这个颜色就是白色）。
- **返回值**：如果函数成功，那么返回值非零；如果函数失败，则返回值为零。

Windows NT：若想获取更多错误信息，请调用GetLastError函数。

**备注**：如果在来源设备信息中可以实行旋转或剪切变换，那么函数BitBlt返回一个错误。如果存在其他变换（并且目标设备信息中匹配变换无效），那么目标设备信息中的矩形区域将在需要时进行拉伸、压缩或旋转。

如果来源和目标设备信息的颜色格式不匹配，那么BitBlt函数将来源场景的颜色格式转换成能与目标格式匹配的格式。当正在记录一个增强型像素文件时，如果来源设备信息标识为一个增强型像素文件设备信息，那么会出现错误。如果来源和目标设备信息代表不同的设备，那么BitBlt函数返回错误。

Windows CE：在Windows CE 1.0版中，参数dwRop只可以指定为下列值：SRCCOPY、SRCAND、SRCPAINT、SRCINVERT。在Windows CE 2.0版中，参数dwRop可以是任何光栅操

作程序代码值。

在这里将使用到这个函数的像素复制功能，即需要设置最后一个参数的SRCCOPY，直接进行复制。另外这里有两个名为HDC类型的参数。在上面的参数解释中，该参数为“指向设备信息的句柄”，如果是初次接触该类型，对这个解释做何理解呢？下面使用一种通俗的语言来描述这个画图过程。

首先，把应用程序的界面想象成一张画板，如果在这个画板上画上纯灰色，就看到一个灰色背景，如果画上一张图片，就可以看到一个图片背景。而在这张画板上画图片的方法，通常是先准备一块和这张画板（A）一样的另一张画板（B）。在画板B上不需要画图，而是只需要把预先准备好的图片（背景图片）贴到这个画板上，当然在程序中这个贴的动作使用API即可完成。在B上贴好图后，再把B画板上的图形复制到A上即可看到程序的背景改变成图形了，这里把B复制到A就要使用上面提到的BitBlt函数，而画板A和B分别对应参数hdcDest和hdcSrc。

由上面内容可以看出，通常情况下需要使用到两块画板A和B，A是已经有了，而B还没有，所以需要创建B。然后把图片贴到B上。

通常情况下使用类似如下代码完成显示图片的功能。

```
//获取程序的界面的HDC(画板A)
HDC hdc = ::GetDC(this->GetSafeHwnd());
//创建一个类似于画板A的画板(画板B)
HDC mdc = CreateCompatibleDC(hdc);
//把图片加载到一个容器bitmap中
CBitmap bitmap;
bitmap.LoadBitmapW(IDB_BACKGROUND);        //IDB_BACKGROUND是刚才加载的图片的ID
//把图片贴到画板B上
HBITMAP oldmdc = (HBITMAP)SelectObject(mdc,bitmap);
//把画板B上内容复制到画板A上
BitBlt(hdc,0,0,200,200,mdc,0,0,SRCCOPY);
//释放资源操作
SelectObject(mdc,oldmdc);
DeleteDC(mdc);
::ReleaseDC(this->GetSafeHwnd(),hdc);
```

只需要几行程序代码就可以实现图片的显示。现在第二个问题出现了，应该把这些实现图片的程序代码放到哪里？答案是，需要把它们放到该程序的WM_PAINT信息处理函数中。当需要画程序的界面时，程序自身就会调用该函数，只需要把这些程序代码放到这个函数中，那么每次都可以显示图片背景。

WM_PAINT信息处理函数在哪里？在程序的项目中，在“Resources View”视图窗口的“Dialog”展开项“IDD_TESTBITMAP_DIALOG”上单击右键，在弹出的快捷菜单中选择“Properties”，弹出该窗体的属性框，在该属性框上单击图3-22标记的按钮。

单击该按钮，在出现的列表左侧找到WM_PAINT，单击该项的右侧，在下拉列表中选择“Add OnPaint”。选择之后，项目将自动添加OnPaint函数的实现程序代码，如图3-23所示。

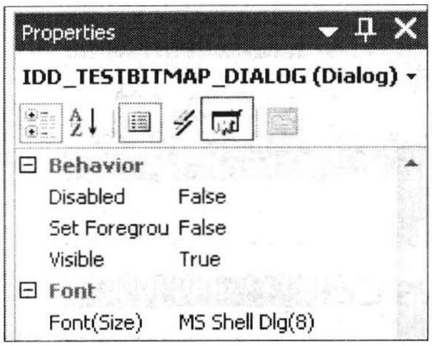



图3-22 按钮的属性对话框

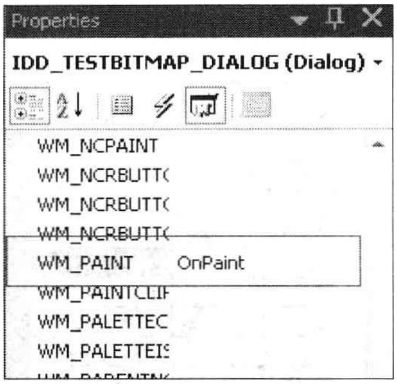



图3-23 添加OnPaint函数的消息映射

在OnPaint函数中添加前面显示图片的程序代码，编译运行后出现类似图3-24所示的界面。至此，已经成功地把背景更改为图片界面。但还有几点不够满意的地方：

- 该程序有一个标题列，不美观。
- 程序没有完全显示图片。

解决方法：

- 去掉标题列，在窗体的属性窗口中把“Border”属性设置成“None”。

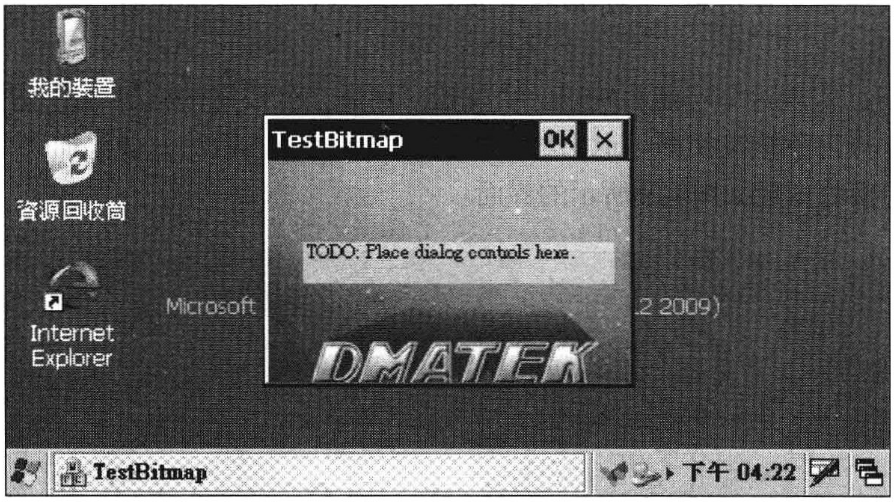



图3-24 程序运行界面

- 把程序的窗口大小设置成200×200像素即可完全显示图片。在OnInitDialog函数中添加如下程序代码即可：

```
SetWindowPos(&wndTop,0,0,200,200,SWP_SHOWWINDOW);
```

OnInitDialog函数在程序启动的时候会自动调用，所以可以把一些初始化程序代码放到这里面。SetWindowPos函数的第一个参数表示设置该窗口的Z坐标轴，第二和第三个参数表示窗口显示在屏幕的左上角的坐标，第四和第五个参数表示窗口的宽和高，最后一个参数SWP_SHOWWINDOW表示把窗口显示出来。

同时删除原来的Label卷标控件。编译修改后的程序，运行结果如图3-25所示。

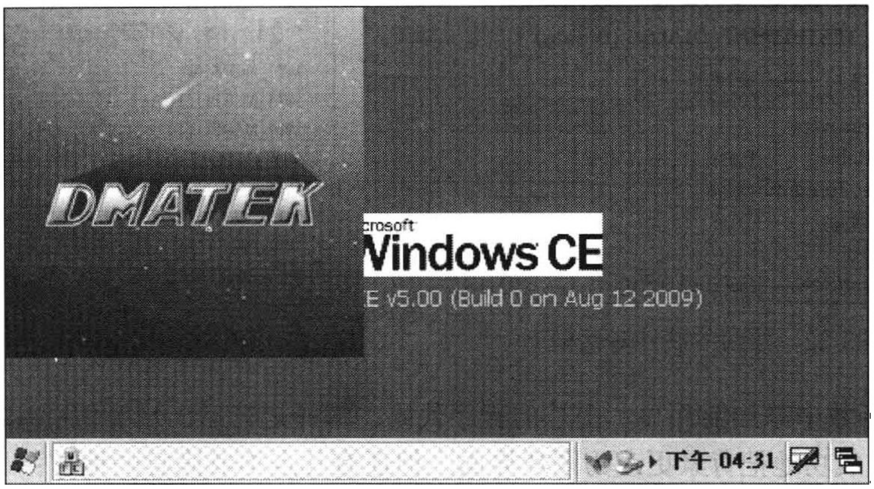



图3-25 美化后的程序运行界面

4. 添加按钮

对于TestBitmap软件，虽然在界面上达到了要求，但是在功能上并没有满足需求，它缺乏最基本的人与软件的交互功能，例如最基本的关闭该软件的功能都没有。下面将介绍如何实现关闭软件的功能。

通常情况下，实现这个功能会使用一个按钮，然后在该按钮中添加退出程序的程序代码。当程序运行后，点一下该按钮即可退出程序。如果按照普通的做法，会添加一个按钮控件，那么此时的软件界面将变成如图3-26所示的界面。

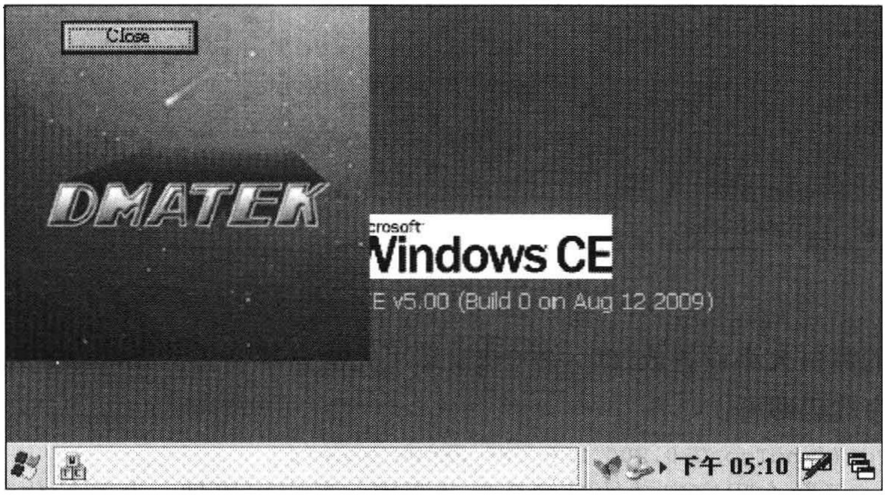



图3-26 给应用程序添加关闭按钮

下面介绍使用检测鼠标信息的方法来模拟按钮的功能。

当鼠标左键在程序界面上按下的时候，程序将收到这个按下的信息WM_LBUTTONDOWN。

和WM_PAINT信息一样，可以实现对该信息的处理函数OnLButtonDown（UINT nFlags，CPoint point），该函数带有两个参数，其中nFlags标示了一些组合信息，这里不讨论它。point标示了当前鼠标按下的点的坐标。这是一个很重要的信息。

当鼠标在程序上进行按下操作时，就会运行该函数，同时在point参数中传递当前鼠标按下处的坐标。

根据上面的描述，下面实现通过单击软件界面来退出程序的功能。那么只需要在OnLButtonDown函数中添加下面这条语句即可：

```
CDialog::OnCancel();
```

这样就使用通过检测鼠标左键的按下信息来仿真一个按钮，只不过这个按钮太大了，有整个应用程序那么大。

下面来细化实现几个小的按钮。

实现的功能如下，当单击图片上的DMATEK这几个字母的时候，会分别显示当前单击的是哪个字母；当单击的不是这几个字母的时候，则会退出程序。

要实现上述功能，就需要先知道DMATEK这几个字母在程序界面上的位置坐标，由于程序界面和图片大小一样，那么也就是这几个字母在图片中的坐标。可以使用Photoshop等类似的图片软件知道它们的坐标分别为：D（21，99，30，27），M（51，99，30，27），A（81，99，20，27），T（101，99，25，27），E（126，99，25，27），K（151，99，27，27）。格式为：字母（左上角X坐标，左上角Y坐标，字母宽度，字母高度）。

在程序中可以使用CRect的一个类来表示这些区域，例如，D的区域表示为Rect_D（21，99，21+30，99+27）。

当在OnLButtonDown添加代码后，则可实现上述功能。如下所示：

```
CRect Rect_D,Rect_M,Rect_A,Rect_T,Rect_E,Rect_K;
Rect_D.SetRect(21,99,21+30,99+27);
Rect_M.SetRect(51,99,51+30,99+27);
Rect_A.SetRect(81,99,81+20,99+27);
Rect_T.SetRect(101,99,101+25,99+27);
Rect_E.SetRect(126,99,126+25,99+27);
Rect_K.SetRect(151,99,151+27,99+27);

if(Rect_D.PtInRect(point))
{
      MessageBox(_T("D"));
}
else if(Rect_M.PtInRect(point))
{
      MessageBox(_T("M"));
}
else if(Rect_A.PtInRect(point))
{
      MessageBox(_T("A"));
}
else if(Rect_T.PtInRect(point))
{
```

```
        MessageBox(_T("T"));
    }
    else if(Rect_E.PtInRect(point))
    {
        MessageBox(_T("E"));
    }
    else if(Rect_K.PtInRect(point))
    {
        MessageBox(_T("K"));
    }
    else
    {
        CDialog::OnCancel();
    }
```

至此，已经介绍了如何给程序添加图片背景，以及使用检测鼠标信息的方式来仿真按钮的方法。需要注意的是，鼠标的信息不仅仅只有左键按下，还有左键释放、滑动，右键按下、释放、滑动等信息。它们的使用方法和左键按下类似。

在打开项目原始程序代码之前，先安装WINCE6.0_CHS_SDK.rar这个SDK。否则可能导致不能打开项目。

## 3.2 窗体基本控件（MFC）

在微软的MFC基础类库提供的控件中，有一些控件是经常使用的，本小节主要介绍常用的基本控件及其使用方法和技巧。由于所有的控件类都是由CWnd类派生出的，因此，控件实际上也是窗口。控件通常是作为对话框的子窗口而创建的，控件也可以出现在视图窗口、工具条和状态条中。控件是独立的小部件，在对话框与用户的交互过程中，担任着主要角色。控件的种类很多，表3-1显示了对话框中的一些基本的控件。

表3-1 MFC常用控件

| 控　件 | 功　能 | 对应控件类 |
|---|---|---|
| 静态文本（StaticText） | 显示正文，一般不能接受输入信息 | CStatic |
| 图片（Picture） | 显示位图、图标、方框和图元文件，一般不能接受输入信息 | CStatic |
| 文本框（EditBox） | 输入并编辑正文，支持单行和多行编辑 | CEdit |
| 命令按钮（PushButton） | 响应用户的输入，触发相应的事件 | CButton |
| 复选框（CheckBox） | 用作选择标记，可以有选中、不选中和不确定三种状态 | CButton |
| 单选按钮（RadioButton） | 用来从两个或多个选项中选中一项 | CButton |
| 组框（GroupBox） | 显示正文和方框，主要用来将相关的一些控件聚成一组 | CStatic |
| 列表框（ListBox） | 显示一个列表，用户可以从该列表中选择一项或多项 | CListBox |
| 组合框（ComboBox） | 是一个编辑框和一个列表框的组合，分为简易式、下拉式和下拉列表式 | CComboBox |
| 滚动条（Scroll Bar） | 主要用来从一个预定义范围值中迅速而有效地选取一个整数值 | CScrollBar |

### 3.2.1 控件的属性、使用方法以及事件

每个控件都有自己的属性、方法以及响应键盘和鼠标操作的事件。学习控件的关键是要注意这些属性、方法和事件的灵活应用。

#### 1. 控件的常用基本属性

属性是指控件的各种性质，例如，控件的ID、控件的显示文本等。属性一般都有默认值，也就是说，在实际编程时即使不设置任何属性值，控件也是可以使用的，只有默认值不能满足要求时才需要更改。对大部分控件来说，有一些常用的属性和事件都是相同的，为避免介绍的内容重复，这里首先将大部分控件共有的属性抽取出来做一些简单介绍，如下所示。

- [ID]属性：用于指定控件的标识符，Windows依靠ID号来区分不同的控件。
- [Caption（标题）]属性：静态文本、组框、按钮、复选框、单选按钮等控件可以显示标题，用来对控件进行文字说明。控件标题中的字符&使紧跟其后的字符有下划线，按Alt+有下划线的字符将启动该控件。若控件是一个单选按钮，则Alt+有下划线的字符将选择该按钮；若是复选框，则相当于对该复选框按空格键；若是按钮，则将激活按钮命令；若控件是一个静态文本，则将激活按Tab顺序紧随其后的下一个控件。
- [Visible]属性：用来指定控件是否可见。
- [Disable]属性：使控件允许或禁止，一个禁止的控件呈灰色显示，不能接收任何输入。
- [Tabstop]属性：用户可以按Tab键移动到具有Tabstop属性的控件上。Tab移动的顺序可以由用户指定。按Ctrl+D或选择菜单“格式”中的Tab键顺序，则Tab顺序会显示出来，用户可以用鼠标来重新指定Tab顺序。默认的Tab顺序是控件的创建先后次序。

#### 2. 控件的使用方法

在Visual C++中使用控件的方法主要有两种：一是为控件添加变量；二是用GetDlgItem函数获取控件的CWnd类型指针。下面的代码是用第二种方法从文本框控件中获取数据。

```
void CDemoDlg::OnBnClickedButton1()
{
//TODO: 在此添加消息处理程序代码
CString str;    //定义一个CString对象,用于保存文本框控件内的文本
//使用GetDlgItem(控件ID)可以获得控件的CWnd类型指针
//通过强制转换,将CWnd*转换成CEdit*
//通过调用GetWindowText方法得到文本框的内容,并保存到str中
((CEdit*t)this->GetDlgItem(IDC_EDIT_TEXT1))->GetWindowTextW(str);
//显示得到的文本
::MessageBox(NULL,str,L" 您在文本框输入了: " ,MB_OK);
}
```

如果采用第一种方法，则需要为文本框控件添加变量。首先右击文本框控件，在弹出的快捷菜单中单击“添加变量”，如图3-27所示。

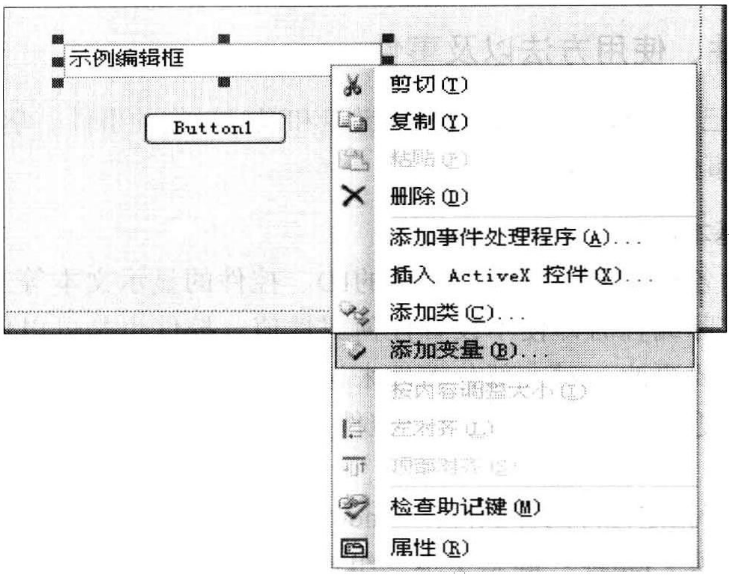



图3-27 为文本框控件添加变量

在弹出的“添加成员变量向导”中，给ID为IDC_EDIT2的控件添加了名为m_CEdit1的变量，单击“完成”按钮结束添加变量向导，如图3-28所示。

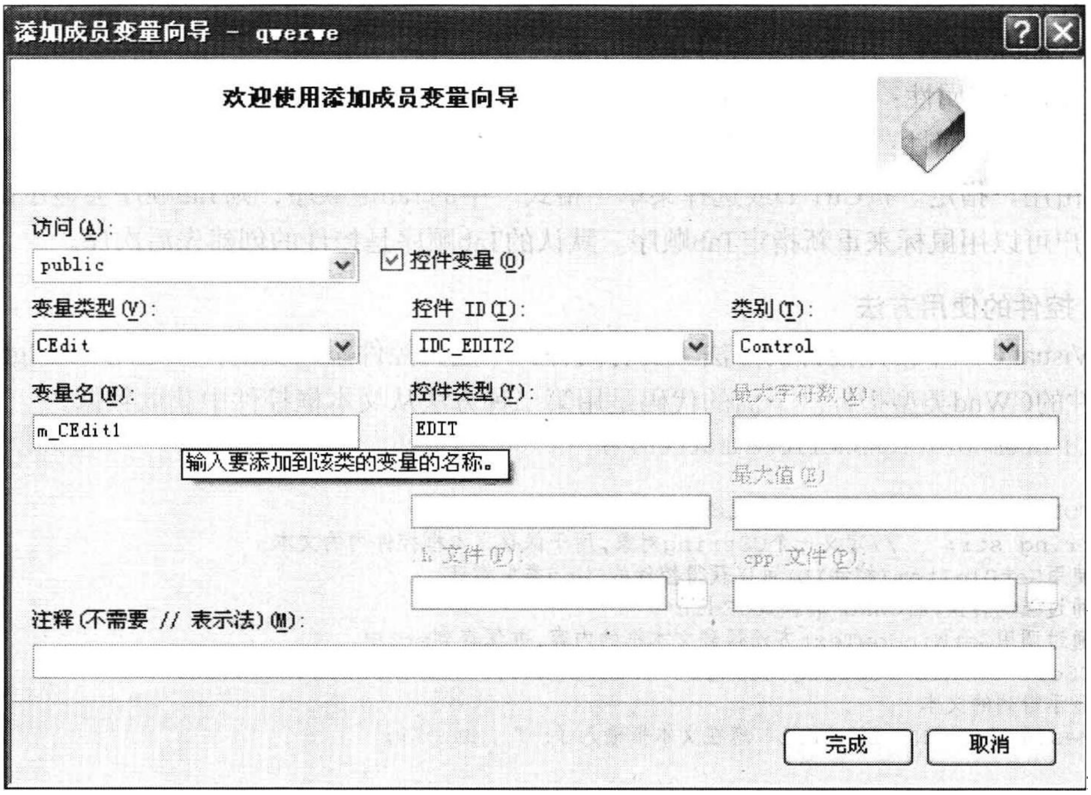



图3-28 添加成员变量向导

此时获取文本框数据的代码如下所示：

```
void CDemoDlg::OnBnClickedButton1()
{
// TODO: 在此添加消息处理程序代码
CString str;    //定义一个CString对象,用于保存文本框控件内的文本
//通过调用GetWindowText方法得到文本框的内容,并保存到str中
m_CEdit1.GetWindowTextW(str);
//显示得到的文本
::MessageBox(NULL,str,L" 您在文本框输入了: " ,MB_OK);
}
```

### 3. 控件的事件

每一种控件对应着多个事件，我们可以为特定的事件添加处理方法，在Visual C++中，用消息来标识事件，每一种事件对应着一个消息，也就是说，事件的发生伴随着消息的产生。下面以按钮控件为例，演示按钮单击事件处理过程的添加步骤。

首先，右击按钮控件，在弹出的菜单中单击“添加事件处理程序”，弹出如图3-29所示“事件处理程序向导”对话框。

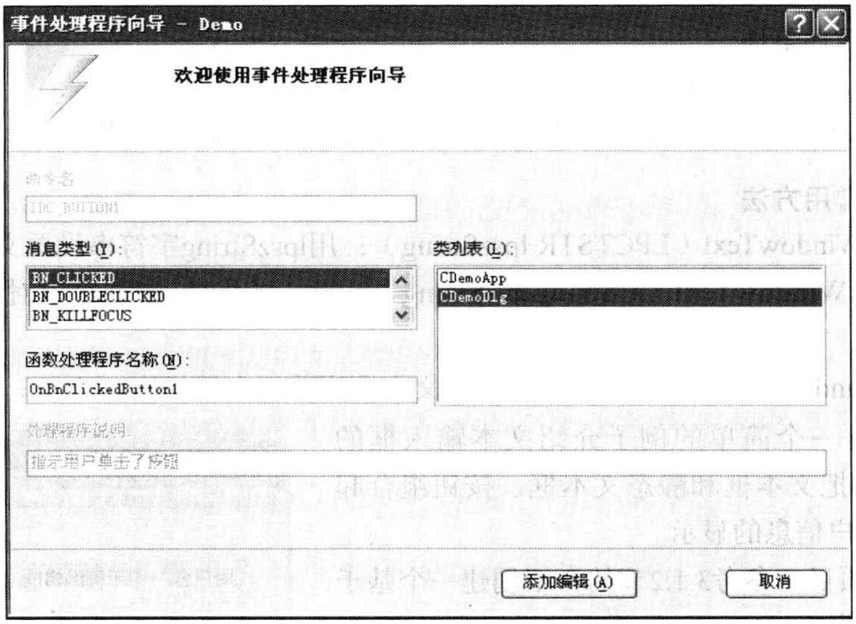



图3-29　添加事件处理程序

在图3-29中，消息类型BN_CLICKED为按钮的单击消息，在右侧“类列表”中选择按钮所在的容器CDemoDlg类，函数处理程序名称为OnBnClickedButton1。单击“添加编辑”按钮结束事件处理程序向导后，直接跳到OnBnClickedButton1方法的代码编辑处，如图3-30所示。

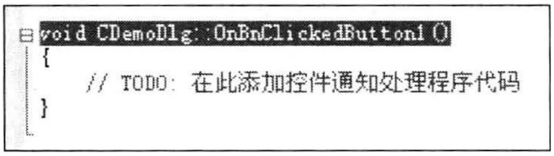



图3-30　为函数处理程序添加代码

## 3.2.2 Edit控件

Edit控件的主要作用是允许用户在应用程序中利用文本框获取或设置文本，当然也可以将控件的只读属性设为True，用于显示文本，而不允许用户编辑文本框中所显示的内容。在文本框中编辑的文本可以是单行的，也可以是多行的，还可以设置为密码字符屏蔽状态作为密码输入框。MFC的CEdit类封装了文本框控件。

### 1. 常用的基本属性

- [Number]属性：只允许将数字输入到文本框控件中。
- [Read Only]属性：防止用户在编辑控件中键入或编辑文本。
- [Multiline]属性：决定文本框控件是否可以包含多行内容。当该属性为False时，只允许输入单行文本；当该属性值为True时，支持输入和显示多行文本。
- [Password]属性：用作密码输入文本框，指定在输入字符时文本框中是否显示代替密码的屏蔽字符星号。如果设置了[Password]属性，即指定了星号作为密码屏蔽字符，则输入的任何字符都显示为该符号。

### 2. 常用的控件事件

- EN_CHANGE消息事件：文本框中的当前内容更改时发出此消息。
- EN_SETFOCUS消息事件：当控件已获得输入焦点时发出此消息。

### 3. 文本框常用方法

- void SetWindowText（LPCTSTR lpszString）；用lpszString字符串设置文本框的内容。
- void GetWindowText（CString& rString）const；用rString字符串对象获取文本框的内容。
- int GetWindowTextLength() const；获取文本框当前文本的长度。

下面将通过一个简单的例子介绍文本输入框的使用方法。该例把文本框和静态文本框、按钮组合起来，用来完成用户信息的显示。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序EditDemo，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置4个StaticText控件、3个Edit控件和一个Button控件，对话框中各控件布局及显示效果如图3-31所示。其中输入密码文本框的Password属性设为True，输入资料文本框设为多行。

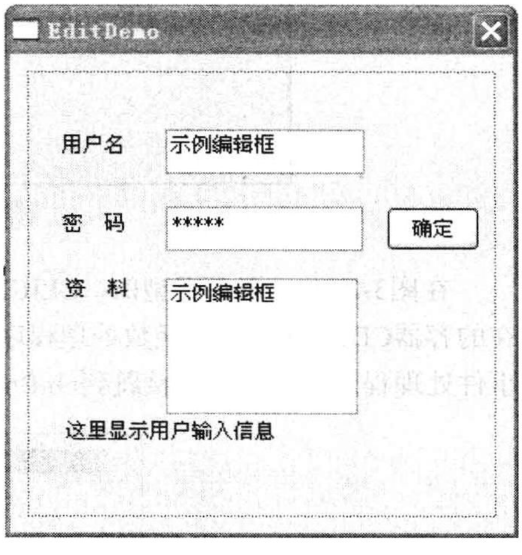



图3-31 文本框使用实例界面设计

对话框上的控件及其属性设置如表3-2所示。

**表3-2 文本框使用实例界面控件及其属性设置**

| 标识（ID） | 描　述 |
|---|---|
| IDC_STATIC1 | 静态文本框，Caption属性值为“资料” |
| IDC_STATIC2 | 静态文本框，Caption属性值为“这里显示用户输入信息”，Visible属性为False |
| IDC_STATIC3 | 静态文本框，Caption属性值为“密码” |
| IDC_STATIC4 | 静态文本框，Caption属性值为“用户名” |
| IDC_EDIT1 | 文本框，用于输入用户名 |
| IDC_EDIT2 | 文本框，用来输入密码，Password属性设为True |
| IDC_EDIT3 | 文本框，输入用户资料，多行文本框 |
| IDC_BUTTON1 | 按钮，Caption属性值为“确定” |

3）在CEditDemoDlg类的OnInitDialog方法里添加如下代码，用于设置文本框控件的初始值。

```
((CEdit *)this->GetDlgItem(IDC_EDIT1))->SetWindowTextW(L"输入用户名");
((CEdit *)this->GetDlgItem(IDC_EDIT2))->SetWindowTextW(L"输入密码");
((CEdit *)this->GetDlgItem(IDC_EDIT3))->SetWindowTextW(L"输入资料");
```

4）为3个文本框的EN_SETFOCUS消息添加消息处理函数，代码如下所示：

```
void CEditDemoDlg::OnEnSetfocusEdit1()
{
((CEdit *)this->GetDlgItem(IDC_EDIT1))->SetWindowTextW(L"");
}
void CEditDemoDlg::OnEnSetfocusEdit2()
{
((CEdit *)this->GetDlgItem(IDC_EDIT2))->SetWindowTextW(L"");
}
void CEditDemoDlg::OnEnSetfocusEdit3()
{
((CEdit *)this->GetDlgItem(IDC_EDIT3))->SetWindowTextW(L"");
}
```

5）为IDC_BUTTON1的BN_CLICKED消息添加消息处理函数，用于显示用户输入的信息，代码如下所示：

```
void CEditDemoDlg::OnBnClickedButton1()
{
CString str1,str2,str3;
((CEdit*)this->GetDlgItem(IDC_EDIT1))->GetWindowTextW(str1);
((CEdit*)this->GetDlgItem(IDC_EDIT2))->GetWindowTextW(str2);
 ((CEdit*)this->GetDlgItem(IDC_EDIT3))->GetWindowTextW(str3);
this->SetDlgItemTextW(IDC_STATIC2,str1+str2+str3);
this->GetDlgItem(IDC_STATIC2)->ShowWindow(true);
}
```

完成以上5个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-32所示。

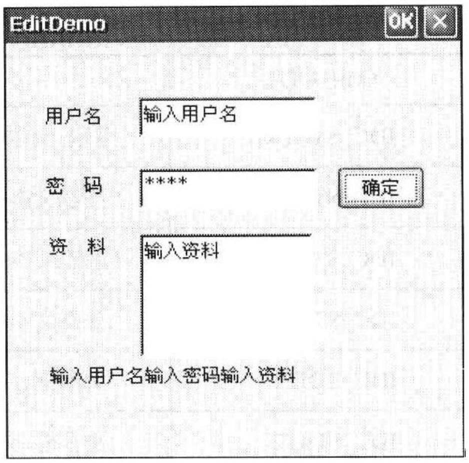


图3-32 文本框使用实例运行效果

## 3.2.3 CheckBox控件

CheckBox控件，即复选框控件，可为用户提供选择功能，常用的是二选一的选择，如“真/假”或“是/否”，但该控件也可以通过属性的设置做三选一的操作。对每一个CheckBox来说，代表的选择都是独立的，若有多个CheckBox控件构成一组选项时，则可以多个同时被选中，相互之间不影响，即复选。MFC的CButton类封装了复选框控件。

### 1. 常用的基本属性

- [Auto]属性：指定复选框在被选中时自动切换状态。
- [Multiline]属性：如果控件文本对控件宽度而言太长，则将文本折为多行。
- [Push Like]属性：指定控件的外观和行为与普通按钮一样。
- [Tri-State]属性：指定复选框为三态。

### 2. 常用的控件事件

- BN_CLICKED消息事件：当用户单击复选框时发出此消息。
- BN_DOUBLECLICKED消息事件：当用户双击复选框时发出此消息。
- BN_SETFOCUS消息事件：当控件已获得焦点时发出此消息。

### 3. 复选框常用方法

- int GetCheck() const；获取复选框的状态，返回值为0表示没有选中，返回值为1表示选中。
- void SetCheck（int nCheck）；设置复选框的状态，nCheck为0表示没有选中，nCheck为1表示选中。

下面通过一个简单的例子来介绍复选框的使用方法，该例把复选框和静态文本框、组框、按钮组合起来，用来完成用户信息的显示。下面介绍该例子的具体实现步骤。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序CheckBoxDemo，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置1个StaticText控件、1个GroupBox控件、3个CheckBox控件和1个Button控件，对话框中各控件布局及显示效果如图3-33所示。

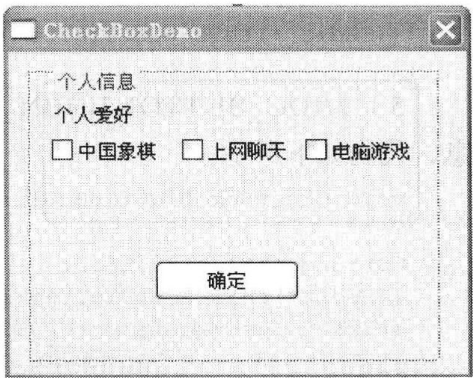


图3-33 复选框使用实例界面设计

**注意：**为了能在Group Box中显示出控件，需要在

Resource编辑界面下按Ctrl + D，对控件进行重新编号，将Group Box的编号改为最大。

对话框上的控件及其属性设置如表3-3所示。

表3-3　复选框使用实例界面控件及其属性设置

| 标识（ID） | 描　述 |
|---|---|
| IDC_STATIC1 | 静态文本框，Caption属性值为"个人爱好" |
| IDC_STATIC2 | 组框，Caption属性值为"个人信息" |
| IDC_CHECK1 | 复选框，Caption属性值为"中国象棋" |
| IDC_CHECK2 | 复选框，Caption属性值为"上网聊天" |
| IDC_CHECK3 | 复选框，Caption属性值为"电脑游戏" |
| IDC_BUTTON1 | 按钮，Caption属性值为"确定" |

3）为IDC_BUTTON1的BN_CLICKED消息添加消息处理函数，用于显示用户输入的信息，代码如下所示：

```
void CCheckBoxDemoDlg::OnBnClickedButton1()
{
CString strMess;
/*复选框也可以关联变量,如果关联的变量值为True表示被选中,此处用另外一种方法。*/
strMess += L"---------------\n你的爱好\n";
CButton * pBtn = NULL ;
pBtn = (CButton * )GetDlgItem(IDC_CHECK1);
if(pBtn&&pBtn->GetCheck()!=NULL)
    strMess += L"中国象棋\n";
pBtn = (CButton * )GetDlgItem(IDC_CHECK2);
if(pBtn&&pBtn->GetCheck()!=NULL)
    strMess += L"上网聊天\n";
pBtn = (CButton * )GetDlgItem(IDC_CHECK3);
if(pBtn&& pBtn->GetCheck()!= NULL)
    strMess += L"电脑游戏\n";
AfxMessageBox(strMess);
}
```

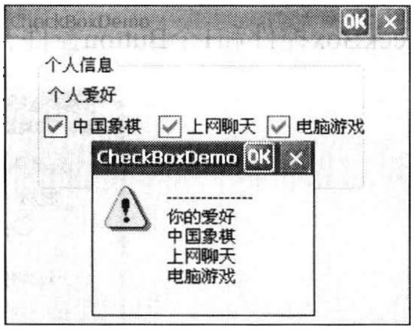



图3-34　复选框使用实例运行效果

完成以上3个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-34所示。

## 3.2.4　RadioButton控件

RadioButton控件，即单选按钮。它与CheckBox控件功能类似，也是用于接收用户的选择，但它是以单项选择的形式出现，即一组RadioButton按钮中只能有一个处于选中状态。一旦某一选项被选中，则同组中其他RadioButton按钮的选中状态自动清除。单选按钮是通过Group属性按Tab键顺序进行分组的。通常情况下为了视图上便于区分，在一个窗体上创建多个单选按钮组时，通常与GroupBox控件或Tab控件一起使用。MFC的CButton类封装了单选按钮控件。

### 1. 常用的基本属性

- [Auto]属性：指定单选按钮在被选中时自动切换状态。
- [Push Like]属性：指定控件的外观和行为与普通按钮一样。

- [Group]属性：指定基于Tab键顺序的一组控件中的第一个控件。
- [Multiline]属性：如果控件文本对控件宽度而言太长，则将文本折为多行。

### 2. 常用的控件事件

- BN_CLICKED消息事件：当用户单击单选按钮时发出此消息。
- BN_DOUBLECLICKED消息事件：当用户双击单选按钮时发出此消息。
- BN_SETFOCUS消息事件：当控件已获得焦点时发出此消息。

### 3. 单选按钮常用方法

- int GetCheck() const；获取单选按钮的状态，返回值为0表示没有选中，返回值为1表示选中。
- void SetCheck（int nCheck）；设置单选按钮的状态，nCheck为0表示没有选中，nCheck为1表示选中。

下面通过一个简单的例子来介绍单选按钮的使用方法，该例在复选框例子的基础上添加了单选按钮，用来完成用户信息的显示。下面介绍该例子的具体实现步骤。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序RadioButtonDemo，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置6个RadioButton控件、4个GroupBox控件、4个CheckBox控件和1个Button控件，对话框中各控件布局及显示效果如图3-35所示。

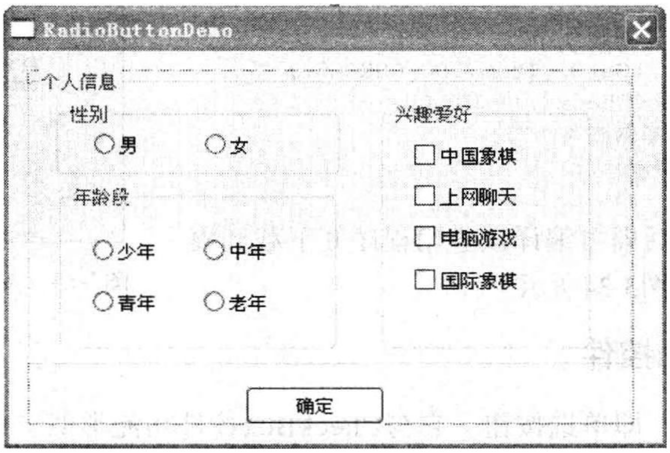



图3-35 单选按钮使用实例界面设计

对话框上的控件及其属性设置如表3-4所示。

表3-4 单选按钮使用实例界面控件及其属性设置

| 标识（ID） | 描 述 |
| --- | --- |
| IDC_RADIO1 | 单选按钮，Caption属性值为“男”，Group属性设置为True |
| IDC_RADIO2 | 单选按钮，Caption属性值为“女” |
| IDC_RADIO3 | 单选按钮，Caption属性值为“少年”，Group属性设置为True |

（续）

| 标识（ID） | 描　述 |
|---|---|
| IDC_RADIO4 | 单选按钮，Caption属性值为“青年” |
| IDC_RADIO5 | 单选按钮，Caption属性值为“中年” |
| IDC_RADIO6 | 单选按钮，Caption属性值为“老年” |
| IDC_STATIC1 | 组框，Caption属性值为“个人信息” |
| IDC_STATIC2 | 组框，Caption属性值为“性别” |
| IDC_STATIC3 | 组框，Caption属性值为“年龄段” |
| IDC_STATIC4 | 组框，Caption属性值为“兴趣爱好” |
| IDC_CHECK1 | 复选框，Caption属性值为“中国象棋” |
| IDC_CHECK2 | 复选框，Caption属性值为“上网聊天” |
| IDC_CHECK3 | 复选框，Caption属性值为“电脑游戏” |
| IDC_CHECK4 | 复选框，Caption属性值为“国际象棋” |
| IDC_BUTTON1 | 按钮，Caption属性值为“确定” |

下面对单选按钮进行分组，分组前所有控件的Tab键顺序如图3-36所示。

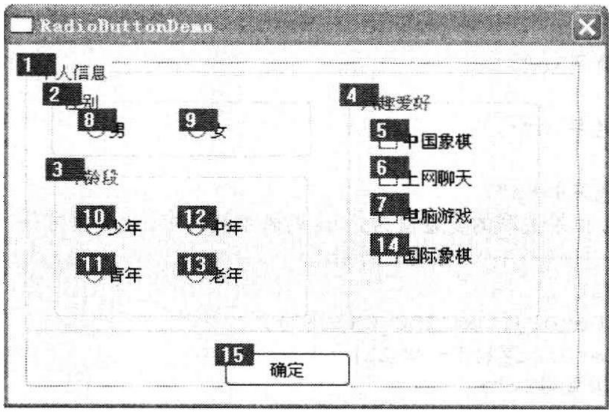



图3-36　分组前控件的Tab顺序

调整后Tab键顺序如图3-37所示。

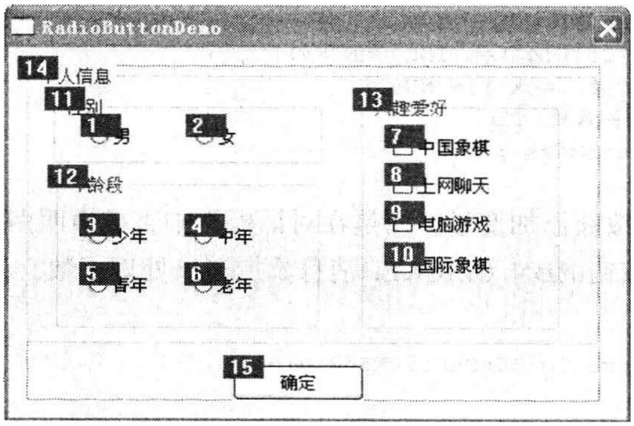



图3-37　重新设定控件的Tab顺序

根据图3-37所示，对单选按钮进行分组，将Tab键顺序值为1和值为3的单选按钮的Group属性设置为True，此时单选按钮被分为性别和年龄段两组。同时为两组单选按钮中的第一个单选按钮添加int类型变量m_sex和m_age。

3）为IDC_BUTTON1的BN_CLICKED消息添加消息处理函数，用于显示用户输入的信息，代码如下所示：

```
void CRadioButtonDemoDlg::OnBnClickedButton1()
{
   CString strMess ;
   UpdateData(TRUE);
   if(m_sex==0)
      strMess = "你是帅哥\n";
   else if(m_sex==1)
      strMess = "你是美女\n";
else
   strMess = "性别未知\n";
if(m_age==0)
   strMess += "你是少年\n";
else if(m_age==1)
   strMess += "你是中年\n";
else if(m_age==2)
   strMess += "你是青年\n";
else if(m_age==3)
   strMess += "你是老年\n";
else
   strMess += "年龄段未知\n";
/*复选框也可以关联变量,如果关联的变量值为True则表示被选中,此处用另外一种方法。*/
strMess += "---------------\n你的爱好\n";
CButton * pBtn = NULL ;
pBtn = (CButton * )GetDlgItem(IDC_CHECK1);
if(pBtn  && pBtn->GetCheck()!= NULL)
    strMess += L"中国象棋\n";
pBtn = (CButton * )GetDlgItem(IDC_CHECK2);
if(pBtn  && pBtn->GetCheck()!= NULL)
strMess += L"上网聊天\n";
    pBtn = (CButton * )GetDlgItem(IDC_CHECK3);
if(pBtn  && pBtn->GetCheck()!= NULL)
    strMess += L"电脑游戏\n";
pBtn = (CButton * )GetDlgItem(IDC_CHECK4);
if(pBtn  && pBtn->GetCheck()!= NULL)
    strMess += L"国际象棋\n";
    AfxMessageBox(strMess);
}
```

4）也可以不为单选按钮添加变量，而是在对话框类中直接声明两个int类型的变量m_sex和m_age。并且为6个单选按钮的BN_CLICKED消息添加消息处理函数，代码如下所示，确定按钮的代码不变。

```
void CRadioButtonDemoDlg::OnBnClickedRadio1()
{
    m_sex=0;
```

```
}

void CRadioButtonDemoDlg::OnBnClickedRadio2()
{
    m_sex=1;
}

void CRadioButtonDemoDlg::OnBnClickedRadio3()
{
    m_age=0;
}

void CRadioButtonDemoDlg::OnBnClickedRadio4()
{
    m_age=1;
}
void CRadioButtonDemoDlg::OnBnClickedRadio5()
{
    m_age=2;
}

void CRadioButtonDemoDlg::OnBnClickedRadio6()
{
    m_age=3;
}
```

完成以上4个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-38所示。

图3-38 单选按钮使用实例运行效果

## 3.2.5 ComboBox控件

ComboBox控件，即下拉文本框，由文本框和列表框两部分组成，也叫组合框。文本框可以用来编辑或者显示当前选中的条目。若列表框隐藏，则单击文本框旁边带有向下箭头的按钮时可弹出列表框，使用键盘或者鼠标可以在列表框中快速选择条目。MFC的CComboBox类封装了组合框控件。

### 1. 常用的基本属性

- [Sort]属性：对添加到列表框的字符串进行自动排序，默认为True。
- [Type]属性：设定组合框的类型。Dropdown类型的组合框既可以手动输入也可以在下拉列表中选择；Drop List类型的组合框只能在下拉列表中选择，不能手动输入。
- [数据]属性：指定用来填充控件的数据，用分号分隔每一项，填充的数据在列表框中显示。用户可以通过此属性填充列表框，但是通常情况下是通过代码动态填充的。

### 2. 常用的控件事件

- CBN_CLOSEUP消息事件：当组合框的列表框关闭时发出此消息。
- CBN_DBLCLK消息事件：当用户双击控件时发出此消息。
- CBN_DROPDOWN消息事件：当组合框的列表框下拉时发出此消息。

### 3. 组合框常用方法

- int GetCount() const；用以获得列表框中的选项数目，失败则返回CB_ERR。
- Int GetCurSel() const；用以得到用户选中下拉列表框中数据的索引值，返回的值是从0开始的，假如没有选择任何选项将会返回CB_ERR（-1）。
- int SetCurSel（int nSelect）；选取列表框中的一个选项。索引从0开始，-1表示什么都不选，当你需要把列表框中已经选取的内容清空时就可以使用这个函数。
- BOOL SetEditSel（intnStartChar，int nEndChar）；设置编辑区中蓝色选中字段的长度，nStartChar是起始的位置，当设置为-1时表示不选择任何字段，nEndChar是结束位置。还有一个对应的函数GetEditSel是取得其位置，在此不详解。
- BOOL LimitText（int nMaxChars）；限制在编辑框中输入的文本长度，当nMaxChars为0时，表示长度最大为65 535字节。
- void GetLBText（int nIndex，CString &rString）；nIndex为列表框中数据的索引值，rString为字符串，其作用是把索引号为nIndex的数据放到rString变量中。
- int GetLBTextLen（int nIndex）；得到索引号为nIndex的数据的长度。
- int AddString（LPCTSTR lpszString）；在列表框的末尾添加一条数据，参数为字符串。
- int DeleteString（UINT nIndex）；删除指定索引为nIndex的数据。
- int InsertString（int nIndex，LPCTSTR lpszString）；在索引nIndex处插入IpszString的内容。
- void ResetContent()；是清除列表框中所有的内容。

下面通过一个简单的例子来介绍组合框的使用方法，该例的“个人信息”区域用来显示用户的姓名、年龄和出生日期，下面的“城市区号邮编”区域可以通过“添加”按钮向复选框的下拉列表中添加数据，通过“修改”按钮修改已有城市的邮编区号信息，通过“退出”按钮退出对话框程序。下面介绍该例子的具体实现步骤。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序ComboBoxDemo，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0

Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置6个ComboBox控件、3个Edit控件、10个StaticText控件、2个GroupBox控件和4个Button控件，对话框中各控件布局及显示效果如图3-39所示。

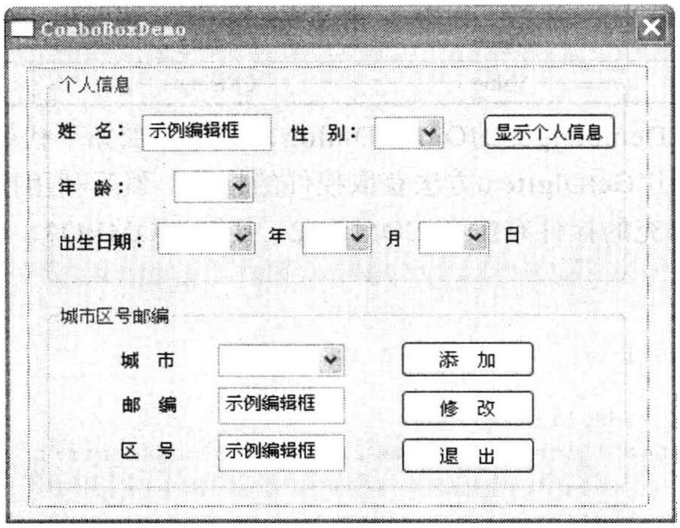



图3-39 组合框使用实例界面设计

对话框上的控件及其属性设置如表3-5所示。

表3-5 组合框使用实例界面控件及其属性设置

| 标识（ID） | 描 述 |
|---|---|
| IDC_COMBO1 | 组合框，“数据”属性设置为“男；女；”，Type属性设为Drop List |
| IDC_COMBO2 | 组合框，Type属性设为Dropdown，输入年龄 |
| IDC_COMBO3 | 组合框，Type属性设为Dropdown，输入年 |
| IDC_COMBO4 | 组合框，Type属性设为Dropdown，输入月 |
| IDC_COMBO5 | 组合框，Type属性设为Dropdown，输入日 |
| IDC_COMBO6 | 组合框，Type属性设为Dropdown，输入城市 |
| IDC_EDIT1 | 文本框，用于输入用户姓名 |
| IDC_EDIT2 | 文本框，用于输入邮编 |
| IDC_EDIT3 | 文本框，用于输入区号 |
| IDC_BUTTON1 | 按钮，Caption属性值为“显示个人信息” |
| IDC_BUTTON2 | 按钮，Caption属性值为“退出” |
| IDC_BUTTON3 | 按钮，Caption属性值为“修改” |
| IDC_BUTTON4 | 按钮，Caption属性值为“添加” |

除了给控件设置属性，还需要给控件添加相应的变量，控件与变量的对应关系如表3-6所示。

表3-6 组合框使用实例控件所对应的变量

| 控件ID号 | 变量类别 | 变量类型 | 变量名 |
|---|---|---|---|
| IDC_COMBO6 | Control | CComboBox | m_ComboBox |
| IDC_COMBO6 | Value | CString | m_strCity |
| IDC_EDIT3 | Value | CString | m_strZone |
| IDC_EDIT2 | Value | CString | m_strZip |

3）在CcomboBoxDemoDlg类的OnInitDialog方法里添加如下代码，用于对组合框控件进行数据填充，首先调用GetDlgItem方法获取控件的指针，然后利用控件的AddString方法对控件进行填充。需要填充的控件有IDC_COMBO2、IDC_COMBO3、IDC_COMBO4、IDC_COMBO5。

```
CString m_str;
for(int i=5;i<=100;i++)
{
    m_str.Format(L"%d",i);
    ((CComboBox*)GetDlgItem(IDC_COMBO2))->AddString(m_str);
}
for(int i=2010;i>=1900;i--)
{
    m_str.Format(L"%d",i);
    ((CComboBox*)GetDlgItem(IDC_COMBO3))->AddString(m_str);
}
for(int i=1;i<=12;i++)
{
    m_str.Format(L"%d",i);
    ((CComboBox*)GetDlgItem(IDC_COMBO4))->AddString(m_str);
}
for(int i=1;i<=31;i++)
{
    m_str.Format(L"%d",i);
    ((CComboBox*)GetDlgItem(IDC_COMBO5))->AddString(m_str);
}
```

4）为IDC_BUTTON1的BN_CLICKED消息添加消息处理函数，用于显示用户输入的信息，代码如下所示：

```
void CComboBoxDemoDlg::OnBnClickedButton1()
{
    CString strMess,m_str;
    strMess += "你的个人信息\n---------------\n";
    ((CEdit*)this->GetDlgItem(IDC_EDIT1))->GetWindowTextW(m_str);
    strMess =strMess+L"姓名: "+m_str+L"\n";
    ((CComboBox*)this->GetDlgItem(IDC_COMBO1))->GetWindowTextW(m_str);
    strMess =strMess+L"性别: "+m_str+L"\n";
    ((CComboBox*)this->GetDlgItem(IDC_COMBO2))->GetWindowTextW(m_str);
    strMess =strMess+L"年龄: "+m_str+L"\n";
    ((CComboBox*)this->GetDlgItem(IDC_COMBO3))->GetWindowTextW(m_str);
    strMess =strMess+L"出生日期: "+m_str+L"年";
    ((CComboBox*)this->GetDlgItem(IDC_COMBO4))->GetWindowTextW(m_str);
    strMess =strMess+m_str+L"月";
```

```
        ((CComboBox*)this->GetDlgItem(IDC_COMBO5))->GetWindowTextW(m_str);
        strMess =strMess+m_str+L"日\n";
        ((CComboBox*)this->GetDlgItem(IDC_COMBO6))->GetWindowTextW(m_str);
        strMess =strMess+L"城市: "+m_str+L"\n" ;
        ((CComboBox*)this->GetDlgItem(IDC_EDIT2))->GetWindowTextW(m_str);
        strMess =strMess+L"邮编: "+m_str+L"\n" ;
        ((CComboBox*)this->GetDlgItem(IDC_EDIT3))->GetWindowTextW(m_str);
        strMess =strMess+L"区号: "+m_str+L"\n" ;
        AfxMessageBox(strMess);
    }
```

5）为CComboBoxDemoDlg类添加IsValidate函数，其返回值类型为bool，此函数用来对输入的城市邮编区号进行完整性检测，代码如下所示：

```
    bool CComboBoxDemoDlg::IsValidate(void)
    {
        UpdateData();
        m_strCity.TrimLeft();
        if(m_strCity.IsEmpty())
    {
            MessageBox(L"城市名输入无效! ");return FALSE;
    }
            m_strZip.TrimLeft();
        if(m_strZip.IsEmpty())
        {
            MessageBox(L"邮政编码输入无效! ");return FALSE;
           }
            m_strZone.TrimLeft();
        if(m_strZone.IsEmpty())
        {
            MessageBox (L"区号输入无效! "); return FALSE;
       }
            return TRUE;
    }
```

6）为IDC_BUTTON4的BN_CLICKED消息添加消息处理函数，此函数用于向IDC_COMBO6控件添加城市信息，同时将邮编和区号保存在CString类型的对象中，并且同组合框中的列表项相关联，达到同时显示的效果，代码如下所示：

```
    void CComboBoxDemoDlg::OnBnClickedButton4()
    {
         if(!IsValidate())return;
         int nIndex = m_ComboBox.FindStringExact( -1,m_strCity);
         if(nIndex != CB_ERR)
         {
            MessageBox(L"该城市已添加! ");
            return;
         }
         nIndex = m_ComboBox.AddString(m_strCity);
         CString strData;
         strData.Format(L"%s,%s",m_strZip,m_strZone);
         //将邮政编码和区号合并为一个字符串
         m_ComboBox.SetItemDataPtr(nIndex,new CString(strData));
```

```
}
```

7）为IDC_BUTTON3的BN_CLICKED消息添加消息处理函数，此函数用于修改已有城市的邮编和区号信息，代码如下所示：

```
void CComboBoxDemoDlg::OnBnClickedButton3()
{
      if(!IsValidate())return;
      int nIndex = m_ComboBox.FindStringExact( -1,m_strCity);
      if(nIndex != CB_ERR)
      {
          delete(CString*)m_ComboBox.GetItemDataPtr(nIndex);
          CString strData;
          strData.Format(L"%s,%s",m_strZip,m_strZone);
          m_ComboBox.SetItemDataPtr(nIndex,new CString(strData) );
      }
}
```

8）为IDC_BUTTON2的BN_CLICKED消息添加消息处理函数，此函数用于退出程序，代码如下所示：

```
void CComboBoxDemoDlg::OnBnClickedButton2()
{
      CDialog::OnCancel();
}
```

9）为对话框的WM_DESTROY消息添加处理函数，此函数用于程序退出前删除所有与列表项相关联的CString数据，并释放内存，代码如下所示：

```
void CComboBoxDemoDlg::OnDestroy()
{
      for(int nIndex = m_ComboBox.GetCount()-1; nIndex>=0; nIndex--)
      {
         //删除所有与列表项相关联的CString数据,并释放内存
         delete(CString *)m_ComboBox.GetItemDataPtr(nIndex);
      }
      CDialog::OnDestroy();
}
```

完成以上9个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-40所示。

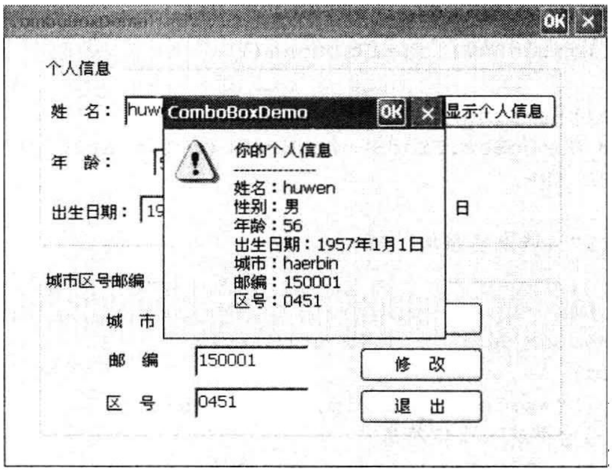



图3-40 组合框使用实例运行效果

## 3.2.6 ListBox控件

ListBox控件，即列表框控件，用于显示一组条目，以便操作者从中选择一条或者多条，然后进行相应的处理，列表框经常用在对话框里，如用列表框选择文件名、目录等。列表框有一个预定义的键盘接口，用户可以用键盘上的箭头和PageUp或PageDown键在列表框中选择数据，或通过适当的样式设置，允许与Shift或Ctrl键组合使用。MFC的CListBox类封装了复选框控件。

### 1. 常用的基本属性

- [MultiColumn]属性：指定水平滚动的多列列表框，默认值为False。
- [Selection]属性：列表框数据的选择模式。取值Single时为单选列表框；取值Multiple时为多选列表框；取值Extended时为扩展多选列表框，允许用户用鼠标拖动或其他特殊组合键进行选择；取值None时为非选列表框，不提供选择功能。
- [Owner draw]属性：自绘列表框，默认为No。
- [Horizontal scroll]属性：选中时，在列表框中创建一个水平滚动条。
- [Vertical scroll]属性：选中时，在列表框中创建一个垂直滚动条。

### 2. 常用的控件事件

- LBN_DBLCLK消息事件：当用户双击控件时发出此消息。
- LBN_SELCHANGE消息事件：列表框中的选择已发生改变时发出此消息。
- LBN_SELCANCLE消息事件：取消列表框中所选内容时发出此消息。

### 3. 列表框常用方法

这些方法用来获得和设置列表框数据的值和属性，所有的CListBox列表框都有这些方法，包括单选列表框、多选列表框和自绘列表框等。

- int GetCount() const；获得列表框中列表项数目。
- int GetCurSel( )const；获得当前单选列表框项的下标（基于0）。
- int SetCurSel（int nSelect）；用来设定某个单选列表项呈选中状态（高亮显示），其中，nSelect指定要设置的列表项索引。
- int GetSelCount( )const；获得多项选择列表框中当前所选的项的数目。
- int GetSelItems（int nMaxItems，LPINT rgIndex）const；将所有当前被选列表框项下标放入一整型数组缓冲区。
- int GetText（int nIndex，LPTSTR lpszBuffer）const；获取选中项的字符串。
- void GetText（int nIndex，CString& rString）const；获取选中项的字符串。
- int AddString（LPCTSTR lpszItem）；向列表框增加列表项。
- int InsertString（int nIndex，LPCTSTR lpszItem）；向列表框的特定位置增加列表项。

下面通过一个较复杂的例子来介绍列表框的使用方法，该例的主要功能为两个列表框的数据交换，以及列表框数据的添加和清除，下面介绍该例子的具体实现步骤。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序ListBoxDemo，编译

环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置2个ListBox控件、1个ComboBox控件、2个StaticText控件、1个GroupBox控件和8个Button控件，对话框中各控件布局及显示效果如图3-41所示。

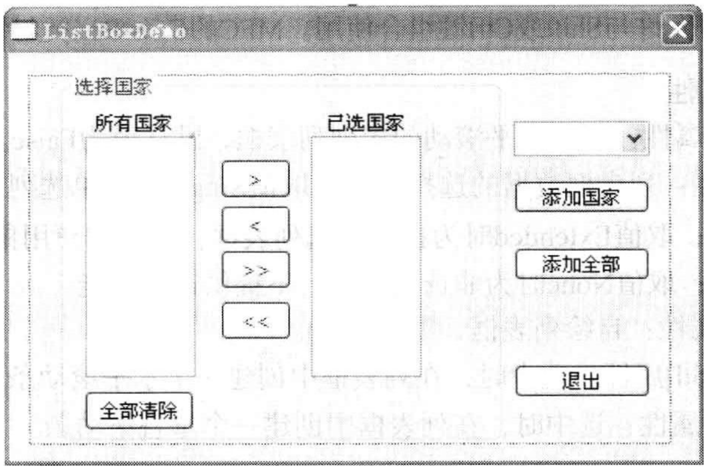



图3-41 列表框使用实例界面设计

对话框上的控件及其属性设置如表3-7所示。

**表3-7 列表框使用实例界面控件及其属性设置**

| 标识（ID） | 描 述 |
|---|---|
| IDC_LIST1 | 左侧列表框，用来显示所有国家。Selection属性设为Single，Sort属性设为False |
| IDC_LIST2 | 右侧列表框，用来显示选择的国家。Selection属性设为Single，Sort属性设为False |
| IDC_COMBO1 | 组合框，“数据”属性设置为“中国；美国；俄罗斯；英国；法国；德国；印度；巴基斯坦；日本；加拿大；澳大利亚；韩国；伊朗；伊拉克；”，Type属性设为Drop List，Sort属性设为False |
| IDC_BUTTON1 | 按钮，Caption属性值为“>” |
| IDC_BUTTON2 | 按钮，Caption属性值为“<” |
| IDC_BUTTON3 | 按钮，Caption属性值为“>>” |
| IDC_BUTTON4 | 按钮，Caption属性值为“<<” |
| IDC_BUTTON5 | 按钮，Caption属性值为“添加国家” |
| IDC_BUTTON6 | 按钮，Caption属性值为“退出” |
| IDC_BUTTON7 | 按钮，Caption属性值为“添加全部” |
| IDC_BUTTON8 | 按钮，Caption属性值为“全部清除” |
| IDC_STATIC1 | 静态文本框，Caption属性值为“所有国家” |
| IDC_STATIC2 | 静态文本框，Caption属性值为“已选国家” |
| IDC_STATIC3 | 静态文本框，Caption属性值为“选择国家” |

除了给控件设置属性，还需要给控件添加相应的变量，控件与变量的对应关系如表3-8所示。

表3-8 列表框使用实例控件所对应的变量

| 控件ID号 | 变量类别 | 变量类型 | 变量名 |
|---|---|---|---|
| IDC_LIST1 | Control | CListBox | m_myList1 |
| IDC_LIST2 | Control | CListBox | m_myList2 |
| IDC_COMBO1 | Control | CComboBox | m_myCombo |

定义一个字符串数组m_myItems，用来存放组合框的项的文本。在对话框类CListBoxDemoDlg中声明，代码如下所示：

```
CString m_myItems[20];
```

IDC_LIST1在初始化时加入了6项。点击“>”按钮就把选中的项移动到IDC_LIST2中去。点击“<”按钮则相反。点击“<<”和“>>”按钮则移动全部项。用“全部清除”按钮可删除IDC_LIST1的全部项。“添加国家”按钮把你选中的在组合框中的项添加到IDC_LIST1中。“添加全部”按钮则把组合框设置的所有项添加到IDC_LIST1中。组合框预置了“中国”这一项，但没有设计删除功能。列表框和组合框类似，它们都有一个从0开始的整数索引和一个用于显示文本的列表。

3）在CListBoxDemoDlg类的OnInitDialog方法里添加如下代码，用于从组合框控件获取项的文本给字符串数组m_myItems赋值。再用m_myItems数组的前六项为IDC_LIST1初始化。最后为组合框设置“中国”为预选项。从组合框控件获取项用函数GetLBText（index，str），它把索引值为index的项的文本复制给str。给列表框赋值用AddString（string）函数，它在列表的尾部增加一项。类似地还可以用插入函数InsertString。

```
int count = m_myCombo.GetCount();  //获取组合框的项数
int i =0;
//从组合框控件获取项的文本
for(i =0; i<count; i++)
     m_myCombo.GetLBText(i,m_myItems[i]);
//List1初始化,六项
for(i =0; i<6 ; i++)
     m_myList1.AddString(m_myItems[i]);
//设置组合框默认选择项为“中国”
m_myCombo.SetCurSel(0);
m_myList1.SetCurSel(0);
this->GetDlgItem(IDC_BUTTON2)->EnableWindow(FALSE);
this->GetDlgItem(IDC_BUTTON4)->EnableWindow(FALSE);
```

4）为IDC_BUTTON1的BN_CLICKED消息添加消息处理函数，用于将选定的数据从IDC_LIST1移动到IDC_LIST2。首先从IDC_LIST1中得到用户选定项的索引号，通过索引号获取选定项的文本，然后删除这一项，再为IDC_LIST2中添加一项。函数GetCurSel可以获取当前选定项索引号。函数GetText（index，item）可以获取指定项的文本，它把指定项的文本复制到字符串item中，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton1()
{
```

```
        CString item;
        int index = m_myList1.GetCurSel();          //获取选定项索引号
        m_myList1.GetText(index,item);              //获取选定项的文本
        m_myList1.DeleteString(index);              //删除一项
        m_myList2.AddString(item);                  //添加到IDC_LIST2中
        m_myList1.SetCurSel(0);                     //设置首项为当前项
        m_myList2.SetCurSel(m_myList2.GetCount()-1);
        if(m_myList1.GetCount()==0){
            his->GetDlgItem(IDC_BUTTON1)->EnableWindow(0);
            this->GetDlgItem(IDC_BUTTON3)->EnableWindow(0);
        }
        this->GetDlgItem(IDC_BUTTON2)->EnableWindow(1);
        this->GetDlgItem(IDC_BUTTON4)->EnableWindow(1);
}
```

5）为IDC_BUTTON2的BN_CLICKED消息添加消息处理函数，用于将选定的数据从IDC_LIST2移动到IDC_LIST1。首先从IDC_LIST2中得到用户选定项的索引号，通过索引号获取选定项的文本，然后删除这一项，再为IDC_LIST1中添加一项，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton2()
{
        CString item;
        int index = m_myList2.GetCurSel();  //获取选定项索引号
        m_myList2.GetText(index,item);      //获取选定项的文本
        m_myList2.DeleteString(index);      //删除一项
        m_myList1.AddString(item);          //添加到IDC_LIST1中
        m_myList2.SetCurSel(0);             //设置首项为当前项
        m_myList1.SetCurSel(m_myList1.GetCount()-1);
        if(m_myList2.GetCount()==0){
            this->GetDlgItem(IDC_BUTTON2)->EnableWindow(0);
        this->GetDlgItem(IDC_BUTTON4)->EnableWindow(0);
        }
        this->GetDlgItem(IDC_BUTTON1)->EnableWindow(1);
       this->GetDlgItem(IDC_BUTTON3)->EnableWindow(1);
}
```

6）为IDC_BUTTON3的BN_CLICKED消息添加消息处理函数，用于将IDC_LIST1中所有数据移动到IDC_LIST2中。采用循环的方式逐条地把数据从IDC_LIST1移动到IDC_LIST2中，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton3()
{
       CString item;
       int index =0;
       while(m_myList1.GetCount() >0){
           m_myList1.GetText(0,item);
           m_myList1.DeleteString(0);
           m_myList2.AddString(item);
       };
m_myList2.SetCurSel(0);
this->GetDlgItem(IDC_BUTTON1)->EnableWindow(0);
this->GetDlgItem(IDC_BUTTON3)->EnableWindow(0);
this->GetDlgItem(IDC_BUTTON2)->EnableWindow(1);
```

```
this->GetDlgItem(IDC_BUTTON4)->EnableWindow(1);
}
```

7）为IDC_BUTTON4的BN_CLICKED消息添加消息处理函数，用于将IDC_LIST2中所有数据移动到IDC_LIST1中。采用循环的方式逐条地把数据从IDC_LIST2移动到IDC_LIST1中，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton4()
{
     CString item;
     int index =0;
     while(m_myList2.GetCount() >0){
        m_myList2.GetText(0,item);
        m_myList2.DeleteString(0);
        m_myList1.AddString(item);
     };
     m_myList1.SetCurSel(0);
     this->GetDlgItem(IDC_BUTTON2)->EnableWindow(0);
     this->GetDlgItem(IDC_BUTTON4)->EnableWindow(0);
     this->GetDlgItem(IDC_BUTTON1)->EnableWindow(1);
     this->GetDlgItem(IDC_BUTTON3)->EnableWindow(1);
}
```

8）为IDC_BUTTON5的BN_CLICKED消息添加消息处理函数，用于将IDC_COMBO1中选定的数据添加到IDC_LIST1中，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton5()
{
     CString item;
     int index = m_myCombo.GetCurSel();  //获取选定项索引号
     m_myCombo.GetLBText(index,item);    //获取选定项的文本
     m_myList1.AddString(item);          //添加到IDC_LIST1中
     m_myList1.SetCurSel(m_myList1.GetCount()-1);        //设置首项为当前项
     this->GetDlgItem(IDC_BUTTON1)->EnableWindow(1);
     this->GetDlgItem(IDC_BUTTON3)->EnableWindow(3);
}
```

9）为IDC_BUTTON6的BN_CLICKED消息添加消息处理函数，用于退出程序，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton6()
{
   OnOK();
}
```

10）为IDC_BUTTON7的BN_CLICKED消息添加消息处理函数，用于将IDC_COMBO1中所有的数据添加到IDC_LIST1中，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton7()
{
     CString item;
     while(m_myList1.GetCount()>0)
     m_myList1.DeleteString(0);
   int n = m_myCombo.GetCount();
   //添加所有项
```

```
    for(int i=0; i<n; i++){
      m_myCombo.GetLBText(i,item);
      m_myList1.AddString(item);
    }
    m_myList1.SetCurSel(0);
    this->GetDlgItem(IDC_BUTTON1)->EnableWindow(1);
    this->GetDlgItem(IDC_BUTTON3)->EnableWindow(1);
}
```

11）为IDC_BUTTON8的BN_CLICKED消息添加消息处理函数，用于清除IDC_LIST1中所有的数据，代码如下所示：

```
void CListBoxDemoDlg::OnBnClickedButton8()
{
    while(m_myList1.GetCount()>0)
        m_myList1.DeleteString(0);
    this->GetDlgItem(IDC_BUTTON1)->EnableWindow(0);
    this->GetDlgItem(IDC_BUTTON3)->EnableWindow(0);
}
```

完成以上11个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-42所示。

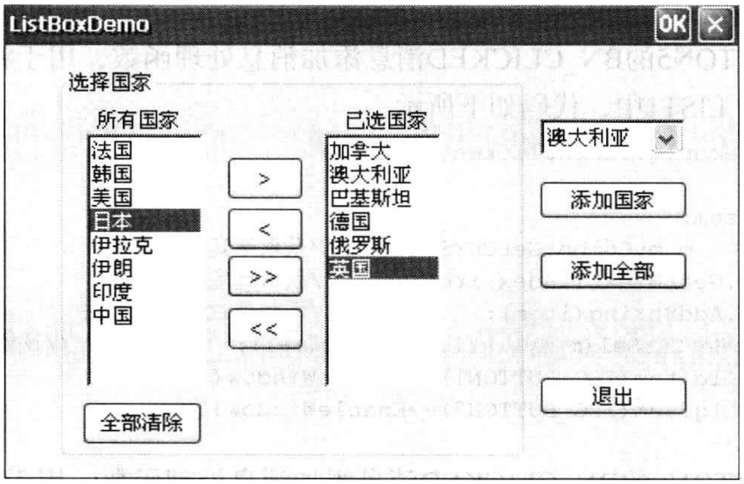



图3-42 列表框使用实例运行效果

## 3.2.7 Spin控件

Spin控件，即旋转按钮控件，是一对箭头按钮。用户可以单击它们来增加或减小某个值，比如一个滚动位置或显示在相应控件中的一个数字。一个旋转控件通常是与一个相伴的控件一起使用的，这个控件称为“伙伴窗口”。一个旋转按钮控件可以自动定位在它的伙伴窗口的旁边，看起来就像一个单一的控件。MFC的CSpinButtonCtrl类封装了旋转按钮控件。

通常，将一个旋转按钮控件与一个编辑控件一起使用，以提示用户进行数字输入。单击向上箭头使当前位置向上限方向移动，而单击向下箭头使当前位置向下限方向移动。默认时，上限为0，下限为100，当用户单击向上箭头则减少数值，而单击向下箭头则增加数值，这看起来就像颠倒一样，因此我们还需使用成员函数CSpinButtonCtrl::SetRange来改变最大值和最小值。

### 1. 常用的基本属性

- [Auto Buddy]属性：自动取伙伴窗口为Tab顺序前一个。
- [Set Buddy Interger]属性：使控件设置伙伴窗口数值，这个值可以是十进制或十六进制。
- [Wrap]属性：数值超过范围时循环。
- [Arrow keys]属性：当按下向上和向下方向键时，控件可以增加或减小。

### 2. 常用的控件事件

- UDN_DELTAPOS消息事件：当控件的位置即将更改时发出此消息。

### 3. 旋转按钮常用方法

- int SetBase（int nBase）；此成员函数用来设置一个旋转按钮控件的基数。这个基数值决定了伙伴窗口显示的数字是十进制的还是十六进制的。十六进制的数是无符号的；十进制的数是有符号的。如果成功则返回先前的基数值；如果给出的是一个无效的基数则返回一个非零值。参数nBase表示控件的新的基数。
- UINT GetBase() const；获取一个旋转按钮控件的基数。
- CWnd* SetBuddy（CWnd* pWndBuddy）；设置伙伴窗口。
- CWnd* GetBuddy() const；获得伙伴窗口。
- int SetPos（int nPos）；用来设置一个旋转按钮控件的当前位置。
- int GetPos() const；获取一个旋转按钮控件的当前位置。
- void SetRange（short nLower，short nUpper）；用来设置一个旋转按钮控件的范围。参数nLower用来设置下限值，参数nUpper用来设置上限值，下限值与上限值无需区分大小，它们只是用来设定数字变化的方向。
- void GetRange（int &lower，int& upper）const；获取一个旋转按钮控件的范围。

下面通过一个简单的例子来介绍旋转按钮和编辑框的配合使用方法，该例把旋转按钮和编辑框组合起来，用来输入RGB的三种颜色，它们的范围都是0～255。以下为该例子的具体实现步骤。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序SpinButtonCtrlDemo，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置3个Spin控件、3个Edit控件、3个StaticText控件、1个GroupBox控件，对话框中各控件布局及显示效果如图3-43所示。

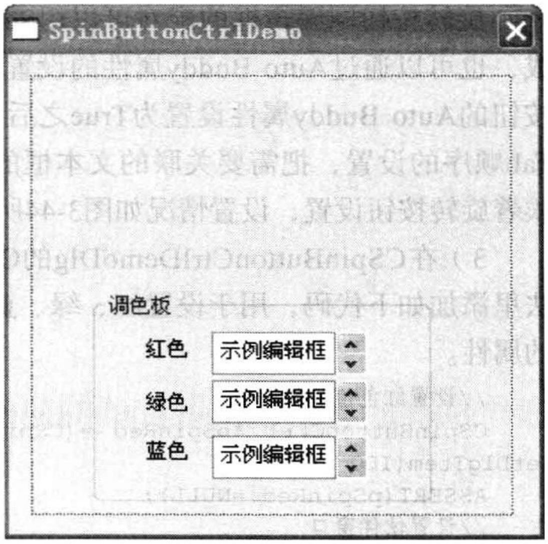



图3-43 旋转按钮使用实例界面设计

对话框上的控件及其属性设置如表3-9所示。

**表3-9 旋转按钮使用实例界面控件及其属性设置**

| 标识（ID） | 描 述 |
|---|---|
| IDC_EDIT1 | 编辑框，用于输入红色值，范围为0～255 |
| IDC_EDIT2 | 编辑框，用于输入绿色值，范围为0～255 |
| IDC_EDIT3 | 编辑框，用于输入蓝色值，范围为0～255 |
| IDC_SPIN1 | 旋转按钮，用于调整红色值，属性Set Buddy Integer设置为True，Auto Buddy属性设置为True |
| IDC_SPIN2 | 旋转按钮，用于调整绿色值。属性Set Buddy Integer设置为True，Auto Buddy属性设置为True |
| IDC_SPIN3 | 旋转按钮，用于调整蓝色值。属性Set Buddy Integer设置为True，Auto Buddy属性设置为True |
| IDC_STATIC1 | 静态文本框，Caption属性值为“红色” |
| IDC_STATIC2 | 静态文本框，Caption属性值为“绿色” |
| IDC_STATIC3 | 静态文本框，Caption属性值为“蓝色” |
| IDC_STATIC4 | 组框，Caption属性值为“调色板” |

除了给控件设置属性之外，还需要给控件添加相应的变量，控件与变量的对应关系如表3-10所示。

**表3-10 旋转按钮使用实例控件所对应的变量**

| 控件ID号 | 变量类别 | 变量类型 | 变量名 |
|---|---|---|---|
| IDC_EDIT1 | VALUE | BYTE | m_red |
| IDC_EDIT2 | VALUE | BYTE | m_green |
| IDC_EDIT3 | VALUE | BYTE | m_blue |

旋转按钮同编辑框的配对可以用SetBuddy方法完成，也可以通过Auto Buddy属性的设置完成，把旋转按钮的Auto Buddy属性设置为True之后，完成对控件Tab顺序的设置，把需要关联的文本框的Tab顺序号紧挨着旋转按钮设置，设置情况如图3-44所示。

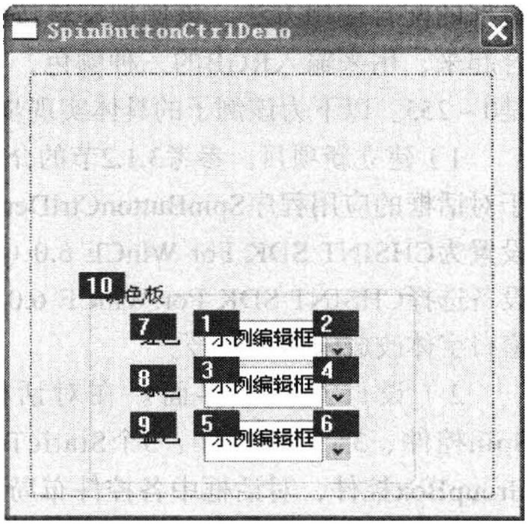



图3-44 控件的Tab顺序设置

3）在CSpinButtonCtrlDemoDlg的OnInitDialog方法里添加如下代码，用于设置红、绿、蓝3个旋转按钮的属性。

```
    //设置红色值旋转按钮
    CSpinButtonCtrl *pSpinRed =(CSpinButtonCtrl*)
GetDlgItem(IDC_SPIN1);
    ASSERT(pSpinRed!=NULL);
    //设置伙伴窗口
    pSpinRed->SetRange(0,255);
    pSpinRed ->SetPos(128);
```

```
//设置绿色值旋转按钮
CSpinButtonCtrl *pSpinGreen=(CSpinButtonCtrl*)GetDlgItem(IDC_SPIN2);
ASSERT(pSpinGreen!=NULL);
//设置伙伴窗口
pSpinGreen->SetRange(0,255);
pSpinGreen->SetPos(128);
//设置蓝色值旋转按钮
CSpinButtonCtrl *pSpinBlue=(CSpinButtonCtrl*)GetDlgItem(IDC_SPIN3);
ASSERT(pSpinBlue!=NULL);
//设置伙伴窗口
pSpinBlue->SetRange(0,255);
pSpinBlue->SetPos(128);
```

4）为CSpinButtonCtrlDemoDlg添加私有方法ChangePaint，用于根据当前的RGB颜色值，绘制具体的图形。ChangePaint方法的实现代码如下所示：

```
void CSpinButtonCtrlDemoDlg:: ChangePaint ()
{
    UpdateData(TRUE);
    CBrush colorBrush;
    COLORREF clRGB;
    //得到RGB颜色值
    clRGB=RGB(m_red,m_green,m_blue);
    CClientDC *pClientDC;
    //得到绘图环境
    pClientDC=new CClientDC(this);
    colorBrush.CreateSolidBrush(clRGB);
    CRect rect(85,10,200,130);
    //显示颜色
    pClientDC->FillRect(rect,&colorBrush);
    delete pClientDC;
}
```

5）为3个颜色的编辑框添加EN_CHANGE事件处理方法，编辑框的此事件处理方法调用ChangePaint方法，实时绘制对应颜色的图形，具体实现代码如下所示：

```
//红色值变化事件
void CSpinButtonCtrlDemoDlg::OnEnChangeEdit1()
{
   ChangePaint ();
   m_tag=true;
}
//绿色值变化事件
void CSpinButtonCtrlDemoDlg::OnEnChangeEdit2() {
   ChangePaint ();
   m_tag=true;
}
//蓝色值变化事件
void CSpinButtonCtrlDemoDlg::OnEnChangeEdit3() {
   ChangePaint ();
   m_tag=true;
}
```

6）由于绘图工作采用的不是响应WM_PAINT消息的OnPaint方法，所以对话框的移动、形

状的改变在触发重绘时，刚刚绘制的图形会消失，解决的办法是添加对话框消息WM_PAINT的处理函数OnPaint，为了在程序第一次启动时不进行重绘操作，在程序中设置一个布尔类型的变量m_tag，初值为False，当程序绘制完图形时将m_tag设置为True，OnPaint方法根据此值来判断是否重绘，代码如下所示：

```
void CSpinButtonCtrlDemoDlg::OnPaint()
{
   if(m_tag==true)
   {
      ChangePaint ();
   }
}
```

完成以上6个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-45所示。

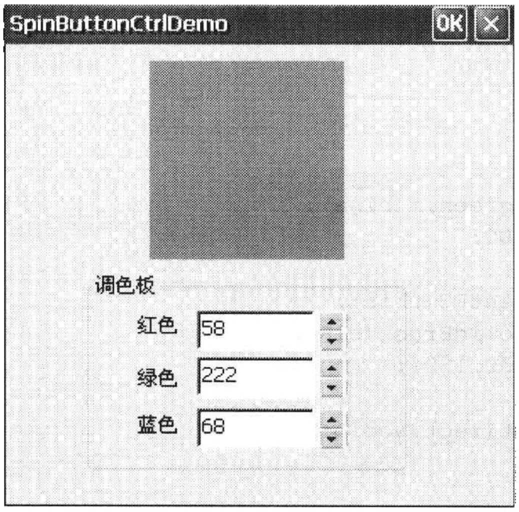



图3-45 旋转按钮使用实例运行效果

### 3.2.8 Slider控件

Slider控件，即滑动条控件，与滚动条相似，滑动条是一种交互式的、直观的控件。它包括一个沿着控件长度滑动的滑块框和用于显示值的范围的可选刻度标志。滑动条控件还有一个内置的键盘界面，也可以使用键盘上的箭头键来移动滑块。

MFC在CSliderCtrl类中提供了Windows滑块常用控件的各项服务，CSliderCtrl也是直接由Cwnd类派生而来，并且继承了CWnd的全部函数。通过编写代码，滑动条控件可以作为Windows的子控件来创建，此外，它还可以被定义用在对话框资源模板中。

#### 1. 常用的基本属性

- [Auto Ticks]属性：指定滑块在其值范围内对于每个增量都有一个刻度线。
- [Orientation]属性：指定滑动条的方向。属性值为Horizontal时表示水平滑动条，取Vertical时为垂直滑动条。
- [Tick Marks]属性：指定滑块显示刻度线。

- [Enable Selection Range]属性：指定滑块只显示选择范围。

### 2. 常用的控件事件

NM_CUSTOMDRAW消息事件：用户自定义绘图消息。

### 3. 滑动条常用方法

- int GetRangeMax() const；获取滑动条的最大值。
- int GetRangeMin() const；获取滑动条的最小值。
- void SetRangeMin（int nMin，BOOL bRedraw = FALSE）；设置滑动条的最小值。
- void SetRangeMax（int nMax，BOOL bRedraw = FALSE）；设置滑动条的最大值。
- int GetPos() const；获取滑动条的当前值。
- void SetPos（int nPos）；设置滑动条的当前值。

下面将通过一个简单的例子来介绍滑动条的使用方法。该例子与旋转按钮例子的功能类似，也将实现设置RGB颜色的功能。下面介绍该例子的具体实现步骤。

1）建立新项目。参考3.1.2节的介绍创建一个基于对话框的应用程序SliderDemo，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）设计对话框界面。在对话框上放置3个Slider控件、1个GroupBox控件、3个StaticText控件，对话框中各控件布局及显示效果如图3-46所示。

对话框上的控件及其属性设置如表3-11所示。

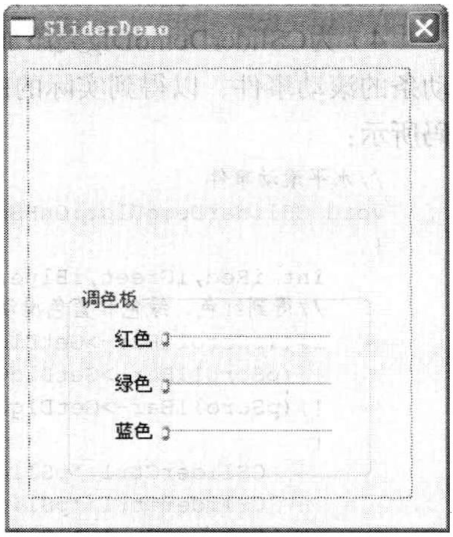



图3-46 滑动条使用实例界面设计

表3-11 滑动条使用实例界面控件及其属性设置

| 标识（ID） | 描 述 |
|---|---|
| IDC_SLIDER1 | 滑动条，用于设置红色值，范围为0 ~ 255 |
| IDC_SLIDER2 | 滑动条，用于设置绿色值，范围为0 ~ 255 |
| IDC_SLIDER3 | 滑动条，用于设置蓝色值，范围为0 ~ 255 |
| IDC_STATIC1 | 静态文本框，Caption属性值为“红色” |
| IDC_STATIC2 | 静态文本框，Caption属性值为“绿色” |
| IDC_STATIC3 | 静态文本框，Caption属性值为“蓝色” |
| IDC_STATIC4 | 组框，Caption属性值为“调色板” |

3）在CSliderDemoDlg类的OnInitDialog方法里添加如下代码，用于滑动条的设置。

```
//设置红色值滑动条
CSliderCtrl *pSliderRed=(CSliderCtrl*)GetDlgItem(IDC_SLIDER1);
ASSERT(pSliderRed!=NULL);
//设置滑动条取值范围
```

```
pSliderRed->SetRange(0,255);
pSliderRed->SetPos(128);
//设置绿色值滑动条
CSliderCtrl *pSliderGreen=(CSliderCtrl*)GetDlgItem(IDC_SLIDER2);
ASSERT(pSliderGreen!=NULL);
//设置滑动条取值范围
pSliderGreen->SetRange(0,255);
pSliderGreen->SetPos(128);
//设置蓝色值滑动条
CSliderCtrl *pSliderBlue=(CSliderCtrl*)GetDlgItem(IDC_SLIDER3);
ASSERT(pSliderBlue!=NULL);
//设置滑动条取值范围
pSliderBlue->SetRange(0,255);
pSliderBlue->SetPos(128);
```

4）为CSliderDemoDlg类添加WM_HSCROLL消息处理方法OnHScroll。该方法将捕捉3个滑动条的滚动事件，以得到实际的RGB颜色值，并绘制具体的图形。OnHScroll方法的实现如下代码所示：

```
//水平滚动事件
void CSliderDemoDlg::OnHScroll(UINT nSBCode,UINT nPos,CScrollBar* pScrollBar)
{
    int iRed,iGreen,iBlue;
    //得到红色、绿色和蓝色滑动条滚动事件
    if((pScrollBar->GetDlgCtrlID()==IDC_SLIDER1)
    ||(pScrollBar->GetDlgCtrlID()==IDC_SLIDER2)
    ||(pScrollBar->GetDlgCtrlID()==IDC_SLIDER3))
    {
        CSliderCtrl *pSliderRed=(CSliderCtrl*)GetDlgItem(IDC_SLIDER1);
        CSliderCtrl *pSliderGreen=(CSliderCtrl*)GetDlgItem(IDC_SLIDER2);
        CSliderCtrl *pSliderBlue=(CSliderCtrl*)GetDlgItem(IDC_SLIDER3);
        iRed=pSliderRed->GetPos();
        iGreen=pSliderGreen->GetPos();
        iBlue=pSliderBlue->GetPos();
    }
    UpdateData(TRUE);
    CBrush colorBrush;
    COLORREF clRGB;
    //得到RGB颜色值
    clRGB=RGB(iRed,iGreen,iBlue);
    CClientDC *pClientDC;
    //得到绘图环境
    pClientDC=new CClientDC(this);
    colorBrush.CreateSolidBrush(clRGB);
    CRect rect(85,10,200,130);
    //显示颜色
    pClientDC->FillRect(rect,&colorBrush);
    delete pClientDC;
    m_tag=true;
    CDialog::OnHScroll(nSBCode,nPos,pScrollBar);
}
```

5）由于绘图工作采用的不是响应WM_PAINT消息的OnPaint方法，所以对话框的移动、形状的改变在触发重绘时，刚刚绘制的图形会消失，解决的办法是添加对话框消息WM_PAINT的

处理函数OnPaint，为了在程序第一次启动时不进行重绘操作，在程序中设置一个布尔类型的变量m_tag，初值为False，当程序绘制完图形时将m_tag设置为True，OnPaint方法根据此值来判断是否重绘，代码如下所示：

```
void CSliderDemoDlg::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    //TODO: 在此处添加消息处理程序代码
    //不为绘图消息调用CDialog::OnPaint()
    if(m_tag==true)
   {
        int iRed,iGreen,iBlue;
        //得到红色、绿色和蓝色滑动条滚动事件
        CSliderCtrl *pSliderRed=(CSliderCtrl*)GetDlgItem(IDC_SLIDER1);
        CSliderCtrl *pSliderGreen=(CSliderCtrl*)GetDlgItem(IDC_SLIDER2);
        CSliderCtrl *pSliderBlue=(CSliderCtrl*)GetDlgItem(IDC_SLIDER3);
        iRed=pSliderRed->GetPos();
        iGreen=pSliderGreen->GetPos();
        iBlue=pSliderBlue->GetPos();
        UpdateData(TRUE);
        CBrush colorBrush;
        COLORREF clRGB;
        //得到RGB颜色值
        clRGB=RGB(iRed,iGreen,iBlue);
        CClientDC *pClientDC;
        //得到绘图环境
        pClientDC=new CClientDC(this);
        colorBrush.CreateSolidBrush(clRGB);
        CRect rect(85,10,200,130);
        //显示颜色
        pClientDC->FillRect(rect,&colorBrush);CSliderCtrl
        delete pClientDC;
        m_tag=true;
   }
}
```

完成以上5个步骤后即可编译该应用程序并下载到模拟器中运行了，效果如图3-47所示。

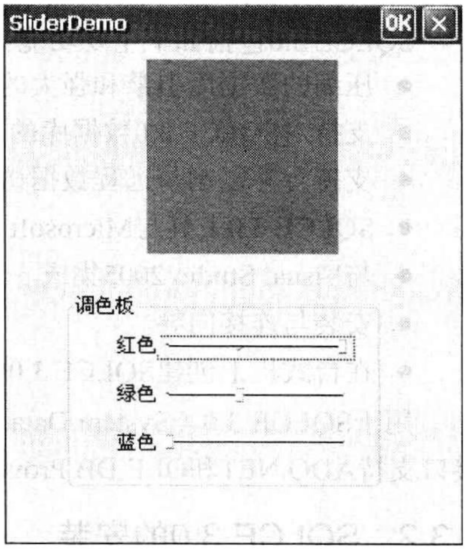



图3-47 滑动条使用实例运行效果图

## 3.3 SQL Server CE

SQL Server CE（SQLCE）为移动智能设备和嵌入式设备提供了一种存取轻量级数据库的解决方案。通过使用Microsoft Visual Studio .NET或者Microsoft eMbedded Visual Tools等开发工具，我们可以将SQL Server的企业数据管理能力扩展到基于Windows CE的智能平台之上。

本节就来介绍SQLCE 3.0数据库的安装及连接配置方法。

### 3.3.1 SQLCE简介

SQL Server CE是一款运行在Windows CE操作系统下的轻量级本地数据库引擎，为移动设

备上偶尔连接的数据访问提供了解决方案。Windows CE开发人员能够使用SQL Server CE开发出将数据管理能力延展到Windows CE移动设备上的应用程序。

虽然SQL Server CE占用的磁盘空间只有3～5MB，但是它在功能上并没有像它的容量那样大幅度缩水。SQL Server CE不但提供了完整的SQL语法，包括内部联接、外部联接和子查询，还内置了数据同步技术的支持，远程数据访问和合并复制确保SQL Server数据库中的数据不仅可靠地提交给用户，而且能够离线修改并在以后的时间里与服务器进行同步，这使得SQLCE成为移动和无线环境的理想选择。有一点要注意的是，SQL Server CE除了可以作为移动数据库在Windows CE设备上运行外，它也支持在PC上存储数据，为桌面应用程序提供数据支持。

SQLCE引擎提供了关系型数据库的基本功能，例如它拥有一个经过优化的查询处理器，并且支持事务和混合数据类型，同时只需占用很少的内存体积和系统资源。

本书将基于SQLCE 3.0版本进行介绍。通过使用Microsoft Visual Studio.NET，SQLCE 3.0和Microsoft.NET Compact Framework可以集成在一起，这样可以简化支持设备的数据库应用程序开发。通过使用SQLCE 3.0数据访问接口管理公共语言运行时的代码，移动应用开发人员可以利用断开连接方案的脱机数据管理能力来生成高度可扩展的应用程序。

SQLCE 3.0包括如下主要功能：

- 压缩的数据库引擎和强大的查询优化器。
- 支持对SQLCE 3.0数据库的多用户访问。
- 支持合并复制与远程数据访问（RDA）。
- SQLCE 3.0工具与Microsoft SQL Server 2005集成。
- 与Visual Studio 2005集成。
- 安装与连接向导。
- 在台式机上创建SQLCE 3.0数据库的功能。

用于SQLCE 3.0（System.Data.SqlServerCe）的Microsoft.NET Compact Framework数据访问接口支持ADO.NET和OLE DB Provider for SQLCE 3.0SQL语法的子集。

## 3.3.2 SQLCE 3.0的安装

在使用SQLCE 3.0之前，先了解一下安装过程。这里包括两个层面的安装，一个层面是客户端环境安装，即在嵌入式电脑或者PDA上安装SQLCE 3.0环境；另外一个层面是服务器环境安装，用于实现服务器同客户端机器的SQLCE数据同步、订阅等工作。如果只是在客户端开发应用，而不需要和服务器进行数据同步等工作，那么可以不用安装服务器环境。

### 1. SQLCE 3.0客户端安装

SQLCE 3.0安装文件可以从微软官方网站下载，也可以从安装Visual Studio 2005之后的目录中获取。在安装Visual Studio 2005时，会包含SQLCE 3.0安装文件，默认目录是C:\ProgramFiles\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0。

在默认目录C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQLServer\Mobile\v3.0\wce500\armv4i下包含了9个.CAB安装文件。这9个安装文件对应3组设备：一组针对

Smart Phone智能手机；一组针对Pocket PC智能PDA；一组针对WINCE 5通用系统。

将sqlce30.ppc.wce5.armv4i.CAB、sqlce30.repl.ppc.wce5.armv4i.CAB和sqlce30.dev.CHS.ppc.wce5.armv4i.CAB这3个文件拷贝到PDA中，并依次安装。

其中sqlce30.ppc.wce5.armv4i.CAB安装文件是SQLCE 3.0核心安装文件；sqlce30.repl.ppc.wce5.armv4i.CAB安装文件用于复制、订阅等相关服务；sqlce30.dev.CHS.ppc.wce5.armv4i.CAB用于安装一个SQLCE查询分析器，便于管理和操作SQLCE，此查询分析器默认目录是\\Program。Files\SQL Mobile\CN\isqlw30.exe。

这3个文件安装完毕，SQLCE 3.0客户端就算安装完毕了。可以通过SQLCE查询分析器来建立SQLCE数据库以及相关数据表，并实现数据存储和查询等相关功能。

### 2. SQLCE 3.0服务器端安装

这里的服务器可以是普通PC，也可以是服务器。操作系统可以是Windows XP SP2、Windows 2000 Server、Windows 2003 Server等。如果SQLCE 3.0应用程序要求在Microsoft SQL Server数据库和SQLCE数据库之间交换数据，则必需在服务器上安装如下软件：

- Microsoft Internet信息服务（IIS）
- SQL Server 2005或SQL Server 2000
- SQLCE 3.0服务器工具

注意：SQLCE 3.0服务器工具的安装文件Sqlce30setupcn.msi默认位于C:\ProgramFiles\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0目录下。安装之后，将会出现“配置Web同步向导”工具。

### 3. 客户端环境和服务器环境体系结构概述

上面介绍了客户端环境和服务器端环境的安装，接下来将重点介绍客户端环境和服务器环境的体系结构，以便对SQLCE 3.0有更深入的了解。

图3-48显示了各种客户端与服务器组件之间的关系。

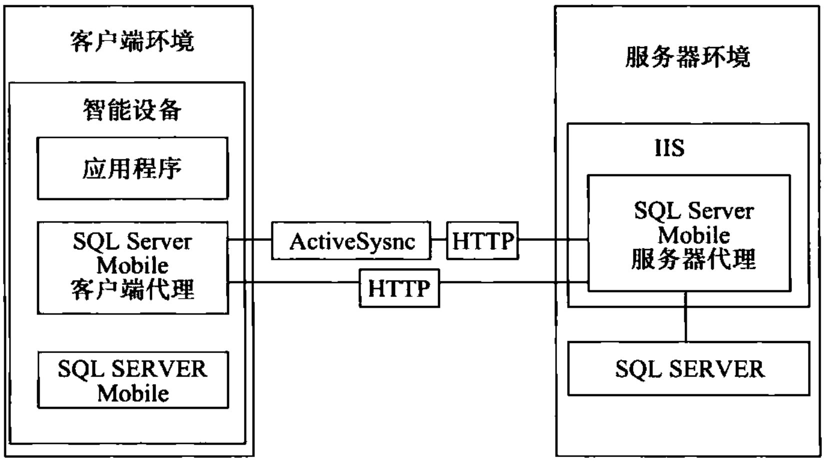



图3-48 SQLCE 3.0体系结构

客户端环境由应用程序与SQLCE组成。

应用程序是采用.NET Compact Framework和Microsoft Visual Studio 2005语言（如Microsoft Visual Basic、Microsoft Visual C#）或者Microsoft Visual C++ for Devices开发的。本书将介绍采用C++开发SQLCE应用程序。

SQLCE中包括用于安装、配置、连接以及数据访问与修改的工具、查询处理器、数据库存储引擎和编程API。其中还包括：

- SQLCE数据库引擎。SQLCE数据库引擎可以管理所支持设备上的SQLCE数据存储区。通过维护每条记录少量的更改跟踪信息，数据库引擎可以跟踪插入、更新或删除的所有数据库记录。在使用复制或远程数据访问（RDA）这两种连接解决方案时，将启用跟踪功能。
- SQLCE客户端代理。SQLCE客户端代理是在支持的设备上进行连接的主要组件。它可以应用下列自定义的SQLCE对象。
- Replication对象
- RemoteDataAccess对象
- Engine对象

通过在应用程序中使用这些对象，可以使用编程方式控制与SQL Server的连接。

服务器环境由SQLCE服务器代理、Microsoft Internet信息服务（IIS）和Microsoft SQL Server组成。

- SQLCE服务器代理可处理由SQLCE客户端代理所发出的HTTP请求。SQLCE服务器代理连接到SQL Server，并通过HTTP将数据与架构信息返回给SQLCE客户端代理。在这步中还会涉及其他的SQL Server连接组件。虽然这些组件未在图3-48中显示，但它们也位于运行IIS的计算机上。
- IIS提供了集成Web服务器功能。各种规模的企业都可以使用IIS托管并管理Internet或Intranet上的网页、FTP站点，还可以通过使用网络新闻传输协议（NNTP）和简单邮件传输协议（SMTP）来路由新闻或邮件。在与SQLCE一起使用时，IIS可以提供一种协议，通过这种协议，设备可以连接到服务器以便通过RDA或复制技术来传输与交换数据。
- Microsoft SQL Server是一组协同工作的组件，它们可以满足规模最大的网站与企业数据处理系统的数据存储与分析需求。其中包括关系数据管理、数据仓库与商务智能组件。SQLCE支持大家熟悉的SQL语法，并提供可以与SQL Server 2000及更高版本一致的开发模型与API。

### 3.3.3 SQLCE 3.0连接配置

在前面的章节中讲述了SQLCE 3.0的安装以及体系结构，本小节以示例的形式介绍如何在客户端和服务器端之间建立代理连接，以便客户端能够和服务器端进行数据交互。

1）在C:\Program Files\Microsoft SQL Server 2005 Mobile Edition\Server目录中找到ConnWiz30.exe可执行程序。双击打开，出现“欢迎使用配置Web同步向导”，如图3-49所示。

图3-49 SQLCE Web同步向导

2）单击“下一步”按钮，进入如图3-50所示的“订阅服务器类型”界面，这里选择SQL Server Mobile Edition。

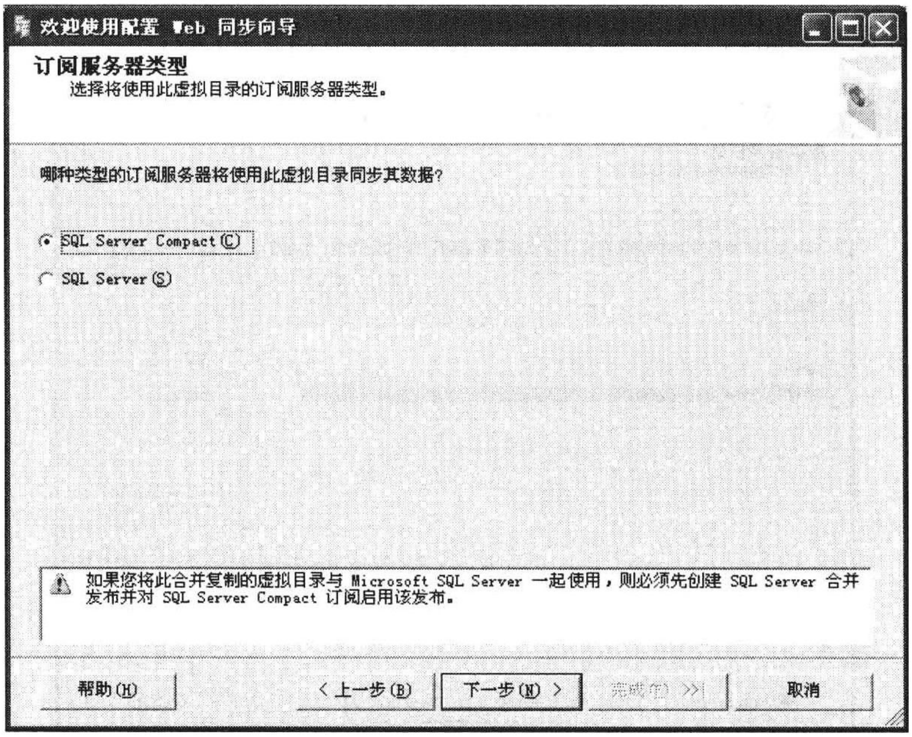



图3-50 订阅服务器类型

3）单击“下一步”按钮，进入设置“Web服务器”界面，这里输入运行IIS的计算机名，采用默认即可，选择“创建新的虚拟目录”单选按钮，选中“默认网站”，具体设置如图3-51所示。

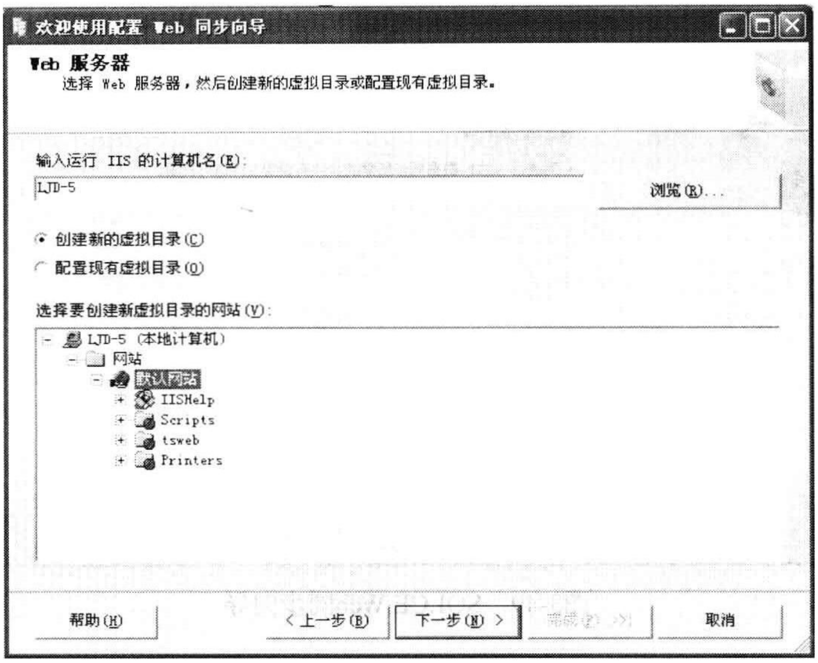



图3-51 Web服务器

4）单击“下一步”按钮，进入如图3-52所示的“虚拟目录信息”界面，这里将别名设置为sqlce30，路径设置为默认。这个默认路径中包含了sqlcesa30.dll文件。

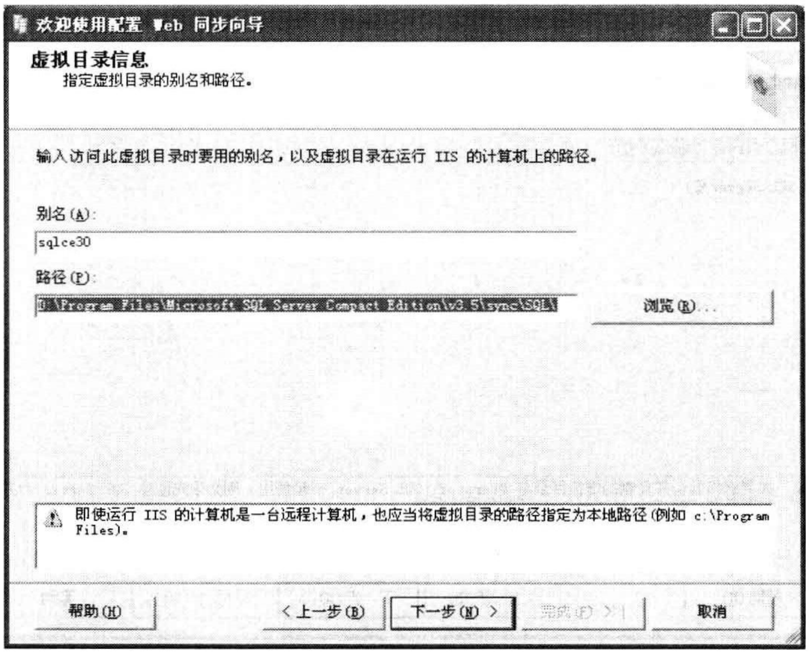



图3-52 虚拟目录信息

5）单击“下一步”按钮，进入“安全通信”界面，选择默认项，如图3-53所示。

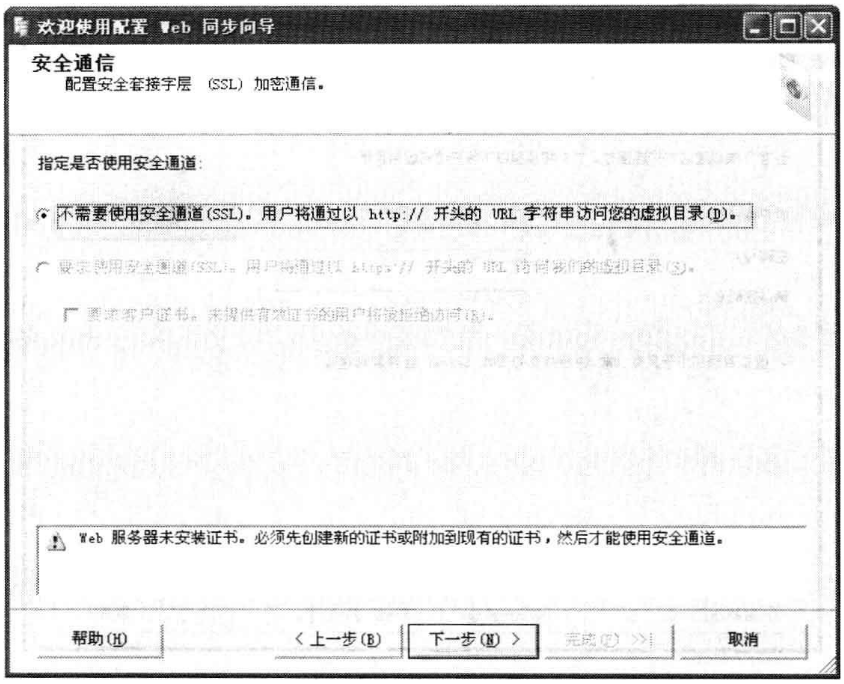



图3-53 安全通信

6）单击“下一步”按钮，进入客户端身份验证界面，该示例中设置为匿名方式，如图3-54所示。

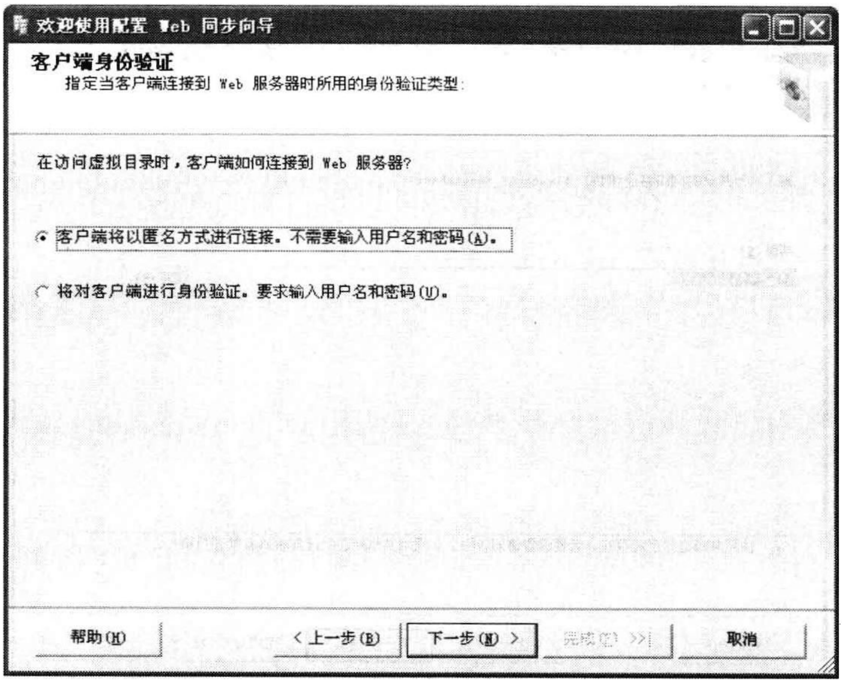



图3-54 客户端身份验证

7）单击“下一步”按钮，设置“匿名访问”参数，选择默认即可，如图3-55所示。

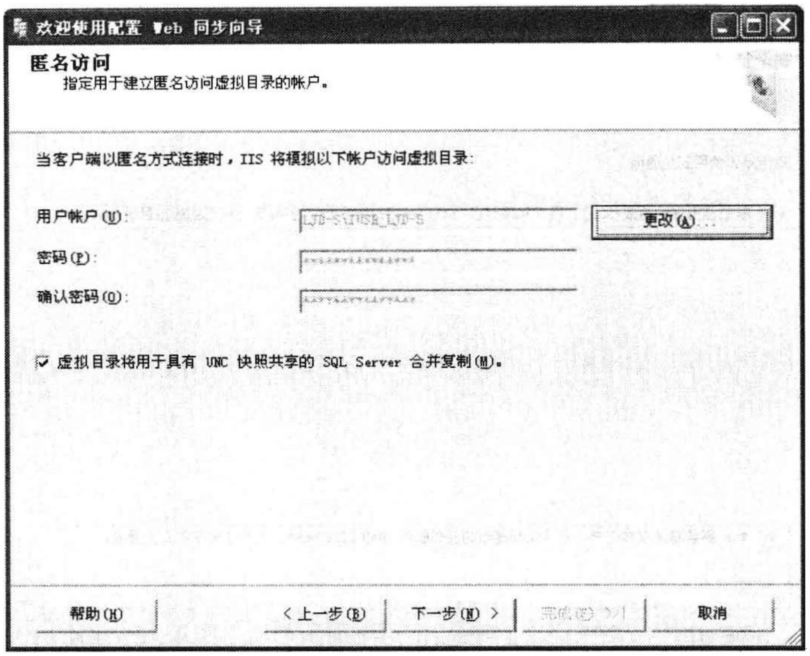



图3-55 匿名访问

8）单击“下一步”按钮，设置“快照共享访问”，这里需要选择一个共享文件目录。本示例在C盘建立了一个新目录snapshot，并将其设置成共享，具备读写权限，共享名为snapshot，因此在共享目录中输入“\\LDJ-5\snapshot”，如图3-56所示。

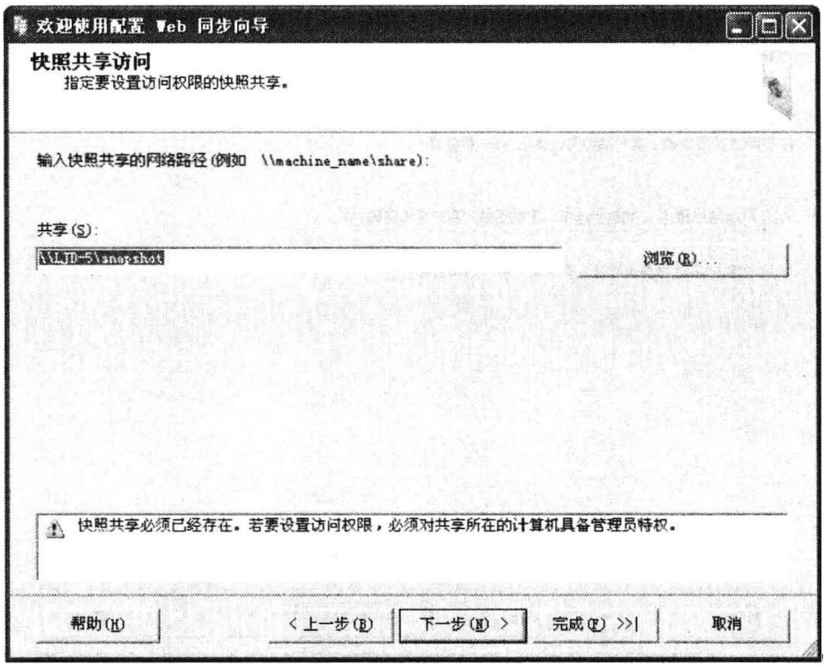



图3-56 快照共享访问

9）单击“下一步”按钮完成配置，配置状态如图3-57所示。

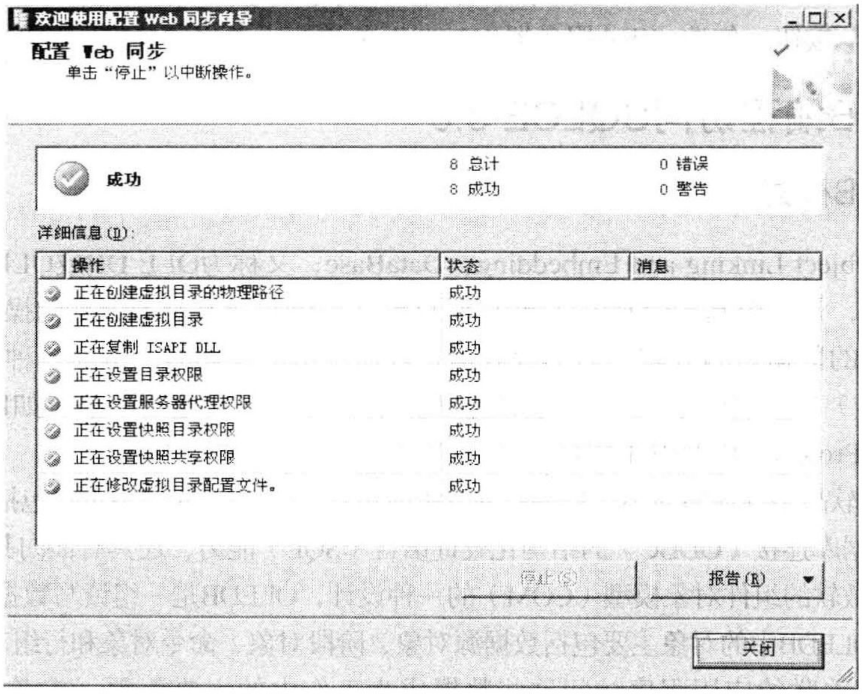



图3-57　Web同步向导配置结束

在SQLCE 3.0连接配置结束后，有一个简单的方法可以判断配置是否成功，即在浏览器中输入SQL Server CE Server Agent（sqlcesa30.dll）的网络路径，当出现如图3-58所示的界面时，表示配置成功。在该例中LJD-5为机器名，sqlce30为虚拟目录别名。

图3-58　测试Web配置状态

**注意**：如果用户使用的是Windows XP SP2操作系统来运行IIS，则必须启用HTTP访问。系

统默认设置为禁用HTTP访问。若要启用HTTP访问，请执行下列操作：

打开“控制面板”→单击Windows防火墙图标→单击“高级”按钮→在“网络连接设置”区域中单击“设置”按钮→勾选“Web服务器(HTTP)”复选框→单击“确定”按钮以确认更改。

# 3.4 OLEDB编程访问SQLCE 3.0

## 3.4.1 OLEDB概述

OLEDB（Object Linking and Embedding，DataBase，又称为OLE DB或OLE-DB，对象链接和嵌入数据库），是一个基于COM的数据存储对象，提供对所有类型数据的操作。通用数据访问提供一种统一的数据访问方式，而不管数据是以何种形式存储的，例如各种不同的文件系统（如FAT或NTFS）、索引顺序文件、个人数据库（如Access）、电子表格（如Excel），项目规划应用程序（如Project）以及电子邮件（如Outlook）。

OLEDB是微软战略性地通向不同数据源的低级应用程序接口。OLEDB不仅包括微软资助的标准数据接口开放数据库连接（ODBC）的结构化查询语言（SQL）能力，还具有面向其他非SQL数据类型的通路。作为微软的组件对象模型（COM）的一种设计，OLEDB是一组读写数据的方法（在过去也称为渠道）。OLEDB中的对象主要包括数据源对象、阶段对象、命令对象和行组对象。

使用多种相关联的应用程序访问这些数据成为工作中的主要瓶颈，或者至少是一件很麻烦的事情。大多数公司都发现自身面临这种情况，他们通过将信息合并到一个数据库管理系统（DBMS）来解决此问题。但是，这种解决方案不仅昂贵、耗时，而且在许多情况下并不可行。另一种解决方案就是开发通用数据访问。OLEDB和ADO提供通用数据访问功能。在这两者中，OLEDB的性能更强一些，并且被推荐和Visual C++应用程序一起使用。

OLEDB为用户提供了一种统一的方法来访问所有不同种类的数据源。OLE DB可以在不同的数据源中进行转换。利用OLEDB，客户端的开发人员在进行数据访问时只需把精力集中在很少的一些细节上，而不必弄懂大量不同数据库的访问协议。OLEDB是一套通过COM接口访问数据的ActiveX接口。这个OLEDB接口相当通用，足以提供一种访问数据的统一手段，而不管存储数据所使用的方法如何。同时，OLEDB还允许开发人员继续利用基础数据库技术的优点，而不必为了利用这些优点而把数据移出来。

由于直接使用OLEDB的对象和接口设计数据库应用程序，因此需要书写大量代码。为了简化程序设计，Visual C++提供了ATL模板用于设计OLEDB数据应用程序和数据提供程序。利用ATL模板可以很容易地将OLEDB与MFC结合起来，使数据库的参数查询等复杂的编程过程得到简化。MFC提供的数据库类使OLEDB的编程更具有面向对象的特性。Viual C++所提供用于OLEDB的ATL模板可分为数据提供程序的模板和数据使用程序的模板。

OLEDB提供了强大高效的功能，但使用C++基于OLEDB来开发数据库应用程序相对ADO.NET来说还是非常复杂的，所以如果用户编写复杂的数据库应用程序，推荐使用ADO.NET进行开发，这样开发效率会高很多。下面将会简要介绍在Windows CE中如何用C++基于OLEDB技术访问SQLCE 3.0数据库。

## 3.4.2 OLEDB编程

OLE DB编程是通过COM方式提供一组接口，应用这些接口可以非常方便地访问SQLCE，具体过程可以参考SQLCE 3.0自带的NorthwindOleDb示例。NorthwindOleDb是一个VisualC++ for Devices示例应用程序，它演示了如何使用OLE DB创建SQL Server 2005 Mobile Edition数据库并与该数据库进行交互。此应用程序通过OLEDB创建数据库、创建表、使用SQL语句插入数据，然后使用IRowsetSeek从数据库中检索数据。

下面简要介绍几个重要的接口并给出示例代码，以使读者对OLEDB编程有直观了解。

### 1. IDBInitialize

IDBInitialize是OLEDB编程用于初始化和释放数据源对象的接口。在进行OLEDB编程时，首先要获得IDBInitialize接口，该接口可通过CoCreateInstance函数创建，示例代码如下：

```
HRESULT hr;
IDBInitialize *pIDBInitialize=NULL;
hr=CoCreateInstance(CLSID_SQLSERVERCE_3_0,0,
                    CLSCTX_INPROC_SERVER,
                    IID_IDBInitialize,
                    (void**)&pIDBInitialize);
```

结合IDBProperties接口设置数据库属性，就可以初始化数据库并建立一个数据库访问会话对象，例如：

```
Const WCHAR DATABASE_NORTHWIND[]=L"Northwindoledb.sdf";
DBPROP Dbprop[1];
DBPROPSET dbpropset[1];
IDBProperties *pIDBProperties=NULL;
VariantInit(&dbprop[0].vValue);
//设置数据库连接属性,即数据库位置
dbprop[0].dwPropertyID=DBPROP_INIT_DATASOURCE;
dbprop[0].dwOptions=DBPROPOPTIONS_ REQUIRED;
dbprop[0].vValue.vt=VT_BSTR;
dbprop[0].vValue .bstrVal= SysAllocString(DATABASE_ NORTHWIND);
if(NULL==dbprop[0].vValue.bstrVal)
{
   hr=E_OUTOFMEMORY;
   goto Exit;
}
//设置属性结合
dbpropset[0].guidPropertySet=DBPROPSET_ DBINIT;
dbpropset[0].rgProperties=dbprop;
dbpropset[0] .cProperties=sizeof(dbprop)/sizeof(dbprop[0]);
//得到IDBProperties接口
hr=pIDBInitialize->QueryInterface(IID_IDBProperties,(void**)&pIDBProperties);
if(FAILED(hr))
   goto Exit;
}
//设置数据库初始化属性
hr=pIDBProperties->SetProperties(1,dbpropset);
if(FAILED(hr))
{
```

```
    goto Exit;
}
//初始化数据库
hr=pIDBInitialize->Initialize();
if(FAILED(hr))
{
    goto Exit;
}
//得到访问数据库的会话对象IDBCreateSession
hr=pIDBInitialize->QueryInterface(IID_ IDBCreateSession,(void**)
&pIDBCreateSession);
```

## 2. IDBDataSourceAdmin

IDBDataSourceAdmin接口用于新建数据库，在SQLCE 3.0中不支持删除和修改数据库的方法，删除数据库时直接删除数据库文件即可。

使用IDBDataSourceAdmin接口创建SQLCE本地数据库的示例代码如下：

```
HRESULT hr=NOERROR;
DBPROPSET dbpropset[2];
DBPROP dbprop[1];//数据库初始化属性
DBPROP sscedbprop[2];//SSCE数据库安全属性
INT i=0;
IDBDataSourceAdmin *pIDBDataSourceAdmin=NULL;
IUnknown *pIUnknownSession=NULL;
//创建OLEDB数据库访问示例,得到IDBDataSourceAdmin接口对象
hr=CoCreateInstance(CLSID_SQLSERVERCE_3_0,0,CLSCTX_INPROC_SERVER,
IID_IDBDataSourceAdmin,(void**)&pIDBDataSourceAdmin);
if(FAILED(hr))
{
    goto Exit
}
VariantInit(&dbprop[0].vValue);
for(int i=0;i<sizeof(sscedbprop)/sizeof(sscedbprop[0]);i++)
{
    VariantInit(&sscedbprop[i].vValue);
}
//设置数据库文件名属性
dbprop[0].dwPropertyID=DBPROP_INIT_DATASOURCE;
dbprop[0].dwOptions=DBPROPOPTIONS_REQUIRED;
dbprop[0].vValue.vt=VT_BSTR;
dbprop[0].vValue.bstrVal=SysAllocString(L"NewDatabase.sdf");
if(NULL==dbprop[0].vValue.bstrVal)
{
    hr=E OUTOFMEMORY;
    goto Exit;
}
//设置数据库需要加密属性
sscedbprop[0].dwPropertyID=DBPROP_SSCE_ENCRYPTDATABASE;
sscedbprop[0].dwOptions=DBPROPOPTIONS_REQUIRED;
sscedbprop[0].vValue.vt=VT_BOOL;
sscedbprop[0].vValue.boolVal=VARIANT_TRUE;
//设置数据库访问密码属性
```

```
Sscedbprop[1].dwPropertyID=DBPROP_SSCE_DBPASSWORD;
Sscedbprop[1].dwOptions = DBPROPOPTIONS_REQUIRED;
Sscedbprop[1].vValue.vt=VT_BSTR;
Sscedbprop[1].vValue.bstrVal=SysAllocString(L"mypassword");
if(NULL==sscedbprop[1].vValue.bstrVal)
{
   hr=E_OUTOFMEMORY;
   goto Exit;
}
//设置创建数据库的属性集
Dbpropset[0].guidPropertySet=DBPROPSET_DBINIT;
Dbpropset[0]. rgproperties = dbprop;
Dbpropset[0] .cProperties=sizeof(dbprop)/sizeof(dbprop[0]);
Dbpropset[1] .guidPropertySet=DBPROPSET_SSCE_DBINIT;
Dbpropset[1] .rgProperties=sscedbprop;
Dbpropset[1] .cProperties=sizeof(sscedbprop)/sizeof(sscedbprop[0]);
//创建新数据库并初始化
hr=pIDBDataSourceA dmin->CreateDataSource(sizeof(dbpropset)/
sizeof(dbpropset[0]),dbpropset,NULL,IID IUnknown, &pIUnknownSession);
if(FAILED(hr))
{
   goto Exit;
}
//至此新数据库已经创建完成
Exit:
//释放相关对象
VariantClear(&dbprop[0].vValue);
VariantClear(&sscedbprop[0].vValue);
VariantClear(&sscedbprop[1].avalue);
if(pIUnknownSession)
{
   pIUnknownSession->Release();
if(pIDBDataSourceAdmin)
{
   pIDBDataSourceAdmin->Release();
}
Return;
```

### 3. IDBCreateSession

IDBCreateSession接口对象用于获取一个新的数据库访问会话对象。在访问数据库时，首先获取一个数据库访问会话对象，然后基于此会话对象执行SQL语句，添加、删除数据以及事务相关工作。在介绍IDBInitialize接口时演示了如何获取IDBCreateSession、删除数据及接口，在介绍下面的相关接口时，会接着演示如何使用IDBCreateSession对象。

### 4. IDBCreateCommand及ICommandText

IDBCreateCommand接口用于创建一个新命令对象，ICommandText表示文本命令对象，例如可以用来执行相关SQL语句。创建命令对象的常规流程是， 根据会话对象得到IDBCreateCommand接口对象，然后利用IDBCreateCommand接口创建一个ICommandText命令对象，最后利用ICommandText文本命令对象执行相关SQL语句。示例代码如下：

```
WCHA RstrSQL[]=L"CREATE TABLE Employees(EmployeeID INT,lastName NVARCHAR(20),FirstName NVARCHAR(10),Address NVARCHAR(60),CityN NVARCHAR(10),Country NVARCHAR(15),Region NVARCHAR(15)PostalCode NVARCHAR(10),Country NVARCHAR(15),HomePhone NVARCHAR(24),Photo IMAGE)";
IDBCreateCommand *pIDBCrtCmd =UNLL;
ICommandText *pICmdText =NULL;
//得到IDBCreateCommand接口对象
hr=pIDBCreateSession->QueryInterface(IID_IDBCreateCommand,(void**)&pIDBCrtCmd);
if(FAILED(hr))
{
   goto Exit;
}
//创建命令对象
hr=pIDBCrtCmd->CreateCommand(NULL,IID_ICommandText,(IUnknown**)&pICmdText);
if(FAILED(hr))
{
   goto Exit;
}
//设置SQL语句
hr=pICmdText->SetCommandText(DBGUID_SQL,strSQL);
if(FAILED(hr))
{
   goto Exit;
}
//执行命令,即执行SQL语句
hr=pICmdText->Execute(NULL,IID_NULL,NULL,NULL,NULL);
```

### 5. ITransactionLocal

在OLEDB中，通过ITransactionLocal接口执行事务功能，事务是通过使用ITransactionLocal::StartTransaction启动的。更改数据库后，可以通过使用ITransactionLocal::Commit提交这些更改，或通过使用ITransactionLocal::Abort中止这些更改。

在SQLCE 3.0中使用事务与在Microsoft SQL Server中使用事务稍有不同，需要特别注意如下3点：

- 不支持ITransactionJoin接口，因为此接口要求访问接口能够支持两个阶段的提交事务。
- SQLCE不支持嵌套事务。
- 在事务完成之前，SQLCE对已在事务中修改的表始终持有排他锁，这意味着DBPROP SUPPORTEDTXNDDL属性的值始终为DBPROPVALes TCse ALL。

下面的代码演示了SQLCE中事务的使用方法。

```
ITransactionLocal   *pITxnLocal  =NULL;
hr=pIDBCreateSession->QueryInterface(IID_ITransactionLocal,(void**)&pITxnLocal);
if(FAILED(hr))
{
   goto Exit;
}
//开始一个新的事务
hr=pITxnLocal->StartTransaction(ISOLATIONLEVEL_READCOMMITTED,0,NULL,NULL);
```

```
if(FAILED(hr))
{
   goto Exit;
}
//执行数据增、删、改等动作
//…
//提交事务
if(pITxnLocal)
{
   pITxnLocal->Commit(FALSE,XACTTC_SYNC,0);
}
goto Exit;
abort:
//回滚事务
if(pITxnLocal)
{
   pITxnLocal->Abort(NULL,FALSE,FALSE);
}
Exit:
//释放资源
```

### 6. IOpenRowset、IRowSet等行集接口

行集是允许OLEDB组件以表格形式显示和修改数据的中心对象。行集对象由若干行组成，其中每一行都包含多列数据。在使用ICommand::Execute返回数据，或者通过直接调用IOpenRowset::OpenRowset都可以创建行集对象。

下列问题在SQLCE中会影响对行集的处理：

- SQLCE支持一次仅检索一行。可以通过调用IRowset::GetNextRows（NULL，0，1）按顺序检索多行，也可以通过在调用IRowset::GetNextRows时指定行号检索特定的行。
- 行集可能可以滚动。此时，可以指定IRowset::GetNextRows（NULL，-1，1）。
- 如果行集所依赖的内部资源由于事务停止而消失，行集可能会进入僵停状态。
- OLEDB Provider for SQLCE支持“通过值”进行数据绑定，而不支持“通过引用”进行数据绑定。
- 插入新行时，SQLCE始终返回计算列的值。这样，客户端就能够立即获取此信息。另外，客户端甚至在向数据存储区提交行之前就可以查看新行的任何标识列值。因此，DBPROP SERVERDATAONINSERT的值必须始终为VARIANT_TRUE。
- 还有一些行集支持的各种接口相对应的其他属性。如果客户端没有通过属性机制来请求要在所生成接口上使用的接口，即使所选择的游标类型支持该功能，这些客户端也不一定能够访问这些接口。

下面以一个代码片段来演示如何通过行集对象向数据库中插入数据，具体代码如下所示：

```
HRESULT hr =NOERROR;                        //函数返回值
DBBINDING *prgBinding =NULL;                //绑定用于创建accessor对象
HROW rghRows[1] ={DB_NULL_HROW};            //存储rowset对象数组
HROW *prghRows =rghRows;                    //Row句柄指针
```

```
DBID TableID;                                 //数据表ID信息,指定表名
DBID IndexID;                                 //指定索引ID
DBPROPSET rowsetpropset[1];                   //打开表的索引属性集
DBPROP rowsetprop[1];                         //打开表的索引属性
ULONG cRowsObtained=0;                        //行集记录个数
DBOBJECT dbObject;                            //DBOBJECT data
DBCOLUMNINFO *pDBColumnInfo =NULL;            //表字段信息
BYTE *pData =NULL;                            //记录缓冲区
WCHAR *pStringsBuffer = NULL;
DWORD dwBindingSize =0;
DWORD dwIndex =0;
DWORD dwRow =0;
DWORD dwCol =0;
DWORD dwOffset =0;
ULONG ulNumCols;
IOpenRowset *pIOpenRowset =NULL;
IRowset *pIRowset =NULL;
ITransactionLocal *pITxnLocal =NULL;
IRowsetChange *pIRowsetChange =NULL;
IAccessor *pIAccessor =NULL;
ISequentialStream *pISequentialStream =NULL;
IColumnsInfo *pIColumnsInfo =NULL;
HACCESSOR hAccessor =DB_NULL_HACCESSOR;
//Accessor handle
VariantInit(&rowsetprop[0].vValue);
//验证IDBCreatesession接口对象
if(NULL==pIDBCreateSession)
{
   hr= E_POINTER
   goto Exit;
}
//创建IOpenRowset对象
hr = m_pIDBCreateSession->CreateSession(NULL,IID_IOpenRowset,
(IUnknown**)&pIOpenRowset);
if(FAILED(hr))
{
   goto Exit;
}
//得到事务对象接口
hr=pIOpenRowset->QueryInterface(IID_ITransactionLocal),
if(FAILED(hr))
{
   goto Exit;
}
//设置打开表的行集属性
TableID.eKind=DBKIND_NAME;
TableID.uName.pwszName=(WCHAR*)TABLE_EMPLOYEE;
IndexID.eKind=DBKIND_NAME;
IndexID.uName.pwszName=L"PK_Employees";
rowsetpropset[0].cProperties=1;
rowsetpropset[0].guidPropertySet=DBPROPSET_ROWSET;
rowsetpropset[0].rgProperties=rowsetprop;
rowsetprop[0].dwPropertyID=DBPROP_IRowsetchange;
```

```
rowsetprop[0].dwOptions=DBPROPOPTIONS_REQUIRED;
rowsetprop[0].colid=DB_NULLID;
rowsetprop[0].vValue.vt=VT_BOOL;
rowsetprop[0].vValue.boolVal=VARIANT_TRUE;

//打开表
hr=pIOpenRowset->OpenRowset(NULL,
                            &TableID,
                            &IndexID,
                            IID IRowset,
                            sizeof(rowsetpropset)/sizeof(rowsetpropset[0])
                            rowsetpropset,
                            (IUnknown**)&pIRowset);
if(FAILED(hr))
{
   goto Exit;
}
//得到IRowsetChange接口
hr=pIRowset->QueryInterface(IID_IRowsetChange,(void**)&pIRowsetChange);
if(FAILED(hr))
{
   goto Exit;
}
//得到IColumnsIof接口,获取字段属性
hr=pIRowset->QueryInterface(IID_IColumnsInfo,(void**)&pIColumnsInfo);
if(FAILED(hr))
{
   goto Exit
}
//获取字段属性
hr=pIColumns Info->GetColumnInfo(&ulNumCols, &pDBColumnInfo, &pStringsBuffer);
if(FAILED(hr)||0==ulNumCols)
{
   goto Exit;
}
//创建DBBINDING数组,不包括bookmark列(第1行)
dwBindingSize=ulNumCols-1;
prgBinding=(DBBINDING*)CoTaskMemAlloc(sizeof(DBBINDING)*dwBindingSize);
if(NULL==prgBinding)
{
   hr =E_OUTOFMEMORY;
   goto Exit;
}
//设置初始绑定位置
dwOffset =0;
//准备绑定数组,以便创建accessor对象
for(dwIndex=0;dwIndex<dwBindingSize; ++dwIndex)
{
  prgBinding[dwIndex].iOrdinal= pDBColumnInfo[dwIndex+1].iordinal;
  prgBinding[dwIndex].pTypeInfo=NULL;
  prgBinding[dwIndex].pBindExt=NULL;
  prgBinding[dwIndex].dwMemOwner=DBMEMOWNER_CLIENTOWNED;
  prgBinding[dwIndex].dwFlags=0;
```

```
  prgBinding[dwIndex].bPrecision=pDBColumnInfo[dwIndex+1].bPrecision;
  prgBinding[dwIndex].bScale=pDBColumnInfo[dwIndex+1].bScale;
  prgBinding[dwIndex].dwPart=DBPART_VALUE|BPART_STATUS|
  DBPART_LENGTH;
  prgBinding[dwIndex].obLength=dwOffset;
  prgBinding[dwIndex].obStatus=prgBinding[dwIndex].obLength+
  sizeof(ULONG);
  prgBinding[dwIndex].obValue=prgBinding[dwIndex].obStatus+
  sizeof(DBSTATUS);
  switch(pDBColumnInfo[dwIndex+1].wType)
  {
    //BLOB对象
    case DBTYPE_BYTES:
      dbObject.dwFlags=STGM_WRITE;
      dbObject.iid=IID_ISequentialStream;
      prgBinding[dwIndex].pObject=&dbObject;
      prgBinding[dwIndex].cbMaxLen=sizeof(IUnknown*);
      prgBinding[dwIndex].wType=DBTYPE_IUNKNOWN;
      break;
      //字符串字段
    case DBTYPE_WSTR:
      prgBinding[dwIndex].pObject=NULL;
      prgBinding[dwIndex].wType=pDBColumnInf[dwIndex+1].wType;
      prgBinding[dwIndex].cbMaxLen=sizeof(WCHAR)*
      (pDBColumnInfo[dwIndex+1].ulColumnSize+1);
      break;
      default:
      prgBinding[dwIndex].pObject=NULL;
      prgBinding[dwIndex].wType=pDBColumnInfo[dwIndex+1].wType;
      gBinding[dwIndex].cbMaxLen=pDBColumnInfo[dwIndex+1].ulColumnSize;
      break;
}
//设置新的偏移量
dwOffset=prgBinding[dwIndex].obValue+prgBinding[dwIndex].cbMaxLen;
//修正8字节对齐
dwOffset=ROUND_UP(dwOffset,COLUMN_ALIGNVAL);
//得到IAccessor接口
hr=pIRowset->QueryInterface(IID_IAccessor,(void**)&pIAccessor);
if(FAILED(hr))
   goto Exit
//创建accessor对象
hr=pIAccessor->CreateAccessor(DBACCESSOR_ROWDATA,
                              dwBindingSize,
                              prgBinding,
                              0,
                              &hAccessor,
                              NULL);
if(FAILED(hr))goto Exit;}
//分配内存,用于存储字段数据
pData=(BYTE*)COTaSkMemAllOC(dwOffset);
if(NULL==pData)
{
   hr=E_OUTOFMEMORY;
```

```
        goto Exit;
    }
    //开始事务
    hr=pITxnLocal->StartTransaction(ISOLATIONLEVEL_READCOMMITTED
    ISOLATIONLEVEL_CURSORSTABILITY,0,NULL,NULL);
    if(FAILED(hr))
    {
        goto Exit;
    }
    //插入数据
    for(dwRow=0; dwRow<sizeof(g_SampleEmployeeData)/sizeof(g-SampleEmployeeData[0]);++dwRow)
    {
        DWORD dwPhotoCol;
        DWORD dwInfoSize;
        LPWSTR lpwszInfo;
        //设置初始偏移量
        memset(pData,0,dwOffset);
        //设置每个字段
        for(dwCol=0;dwCol<dwBindingSize;++dwCol)
        {
            //得到要插入字段对应的值
            lpwszInfo=g_SampleEmployeeData[dwRow].wszEmployeeInfo[dwCol];
            switch(prgBinding[dwCol].wType)
            {
              //字符串
              case DBTYPE_WSTR:
              dwInfoSize=prgBinding[dwCol].cbMaxLen/sizeof(WCHAR)-1;
              if(wcslen(lpwszInfo)>=dwInfoSize)
              {
                wcsncpy((WCHAR*)(pData+prgBinding[dwCol].obValue),lpwszInfo,dwInfoSize);
               *(WCHAR*)(pData+prgBinding[dwCol].obValue+dwInfoSize*sizeof(WCHAR))=WCH
                 AR('\0');
               }
               else
               {
                wcscpy((WCHAR*)(pData+prgBinding[dwCol].obValue),lpwszInfo);}
               *(ULONG*)(pData+prgBinding[dwCol].obLength)=wcslen((WCHAR*)
                (pData+prgBinding[dwCol].obValue))*sizeof(WCHAR);
                *(DBSTATUS*)(pData+prgBinding[dwCol].obStatus)= DBSTATUS_S_OK;
                break;
              case DBTYPE_IUNKNOWN:
                dwPhotoCol=dwCol;
              break;
              //整数
              case DBTYPEI_4:
                  *(int*)(pData+prgBinding[dwCol].obValue)=
                  _wtoi(g_SampleEmployeeData[dwRow].wszEmployeeInfo[dwCol]);
                  *(ULONG*)(p Data+prgBinding[dwCol].obLength)=4;
                  *(DBSTATUS*)(pData+prgBinding[dwCol] .obStatus)= DBSTATUS_S_OK;
                  break;
              default:
              break;
        }
```

```
}
//插入一行数据
hr=pIRowsetChange->InsertRow(DB_NULL_ HCHAPTER,hAccessor,pData,
prghRows);
if(FAILED(hr))
{
   goto Abort;
}
//得到插入的行数据
hr=pIRowset->GetData(rghRows[0],hAccessor,pData);
if(FAILED(hr))
{
   goto Abort;
}
//检查状态
if(DBSTATUS_S_OK!=*(DBSTATUS*)(pData+prgBinding[dwPhotoCol].obStatus))
{
   hr=E_FAIL;
   goto Abort;
}
//通过ISequentialStream接口插入雇员图片字段
pISequentialstream=(*(ISequentialStream**)(pData+
prgBinding[dwPhotoCol].obValue));
if(pISequentialstream)
{
   //插入照片
   hr=SaveEmployeePhoto(pISequentialStream,
   g_SampleEmployeeData[dwRow].dwEmployeePhoto);
if(FAILED(hr))
{
   goto Abort;
   //释放ISequentialStream接口
   hr=pISequentialstream->Release();
if(FAILED(hr))
{
   pISequentialstream=NULL
   gotoAbort;
}
pISequentialStream=NULL;
//释放rowset对象
hr=pIRowset->ReleaseRows(1,prghRows,NULL,NULL,NULL);
if(FAILED(hr))
{
   goto Abort;
}
prghRows[0]=DB_NULL_HROW;
}
//Commit the transaction
//提交数据
if(pITxnLocal)
{
```

```
    pITxnLocal->Commit(FALSE,XACTTC SYNC,0);
}
goto Exit
Abort:
if(DB NULL HROW!=prghRows(0))
{
    pIRowset->ReleaseRows(1,prghRows,NULL,NULL,NULL);
    //回滚事务,取消插入
if(pITxnLocal)
{
    pITxnLocal->Abort(NULL,FALSE,FALSE);
    Exit:
    //释放资源
return hr;
```

## 3.5 SQLCE远程数据访问

远程数据访问，即Remote Data Access（RDA），为Windows CE移动设备上SQLCE与SQL Server数据库之间的同步提供了快速而简便的解决方案。通过使用这项技术，并利用各种网络环境，用户可以将PDA连接到远程的SQL Server 2000/2003数据库。例如，工商管理人员在执行巡查任务前，可以用Windows CE移动设备把台式计算机中的数据库（SQL Server）复制到Windows CE的数据库（SQLCE）上。在执行任务时，可以在移动设备上修改记录。完成任务后，可以把移动设备上的数据库上传到台式计算机中的数据库，使这两个数据库之前的数据保持同步。

通过RDA技术，可以完成下面几种类型的数据传递：

- 从服务器端下载数据。当移动设备上的本地数据库无内容时，可以通过RDA技术从远程SQL Server 2000服务器下载初始化数据到本地SQLCE数据库。例如，应用程序利用RDA技术下载终端用户登录数据和产品原始数据到本地移动数据库中，从而完成系统初始化。如果没有这个最初的初始化步骤，系统将无法登录。后面的实例讲解中将详细描述这一过程。
- 向服务器上传数据。移动设备上的应用程序在初始化后，能够让操作员登录并使用系统。操作员录入和采集的信息（如GPS的信息或条形码的信息）都将被保存在移动数据库中，此时，又可以利用RDA技术将本地数据上传到SQL Server 2000服务器端。值得注意的是，在设计服务器端数据库时，在参加同步的表中不能存在自增量字段，否则容易造成上传数据失败，其原因是不同的移动设备会对表中同一个字段做自增。
- 更新服务器端数据。有的时候，在多个移动设备上的数据库和服务器端的数据库中都可能存在同一条记录，这时服务器端的记录将以最后一次同步移动设备中的记录为准。
- 下达无返回的T-SQL命令操作远程服务器（Submit SQL）。移动设备上的应用程序能够利用T-SQL命令远程操作SQL Server 2000数据库。这项功能非常实用。通常需要在采集数据时记录当前时间，但是如果存在多个移动设备，各个移动设备的机器时间又可能是不统一的，对这种情况的解决方案就是在每次数据同步时，利用Submit SQL技术刷新

服务器时间，下载这个时间并同时设定为移动设备的时间。

更多信息可以参考SQL Server CE简介中的SQL Server CE架构图。接下来将介绍SQLCE提供的“拉数据”、“推数据”和“远程T -SQL操作”3种方法。

## 3.5.1 拉数据

拉数据（Pull）就是Windows CE从SQL Server服务器获取数据的过程，这里也可以理解为从服务器中下载数据。用户不必关心这之间的处理过程，只需要调用ISSCERDA接口的Pull方法即可。Pull方法的定义如下：

```
virtual/*[id]*/HRESULT STDMETHODCALLTYPE Pull(
/*[in]*/BSTR LocalTableName,
/*[in]*/BSTR SQLSelectString,
/*[in]*/BSTR OLEDBConnectionString,
/*[defaultvalue][in]*/RDA TRACKOPTION TrackOption,
/*[defaultvalue][in]*/BSTR ErrorTableName)=0;
```

在上述代码中，各参数的具体含义如下：

- 参数LocalTableName表示将从服务器获取的数据存储到此表中。如果此表已经存在，将会发生错误。
- 参数SQLSelectString表示从服务器获取数据的SQL表达式。SQLSelectString将控制从服务器取得什么数据。例如，如果SQLSelectString为SELECT * FROM Customers WHERE State='CA'，那么将只能获取那些居住在加利福尼亚州的客户信息。同时SQLSelectString也支持那些返回结果集的存储过程和视图。如果是将服务器中多个表的数据传递到CE中的一个表，此时应将TrackOption参数值设置为TRACKINGOFF，表示获取后的数据不能再上传到服务器。
- 参数OLEDBConnectionString表示连接到SQL Server服务器的连接字符串。
- 参数TrackOption表示SQLCE是否跟踪存储获取服务器数据的表（即LocalTableName）的变化。

参数TrackOption的取值是一个枚举值，具有如表3-12所示的4个取值。

表3-12 参数TrackOption取值表

| 常 量 | 值 | 描 述 |
|---|---|---|
| TRACKINGON | 1 | 对于接收到数据的表，SQLCE将跟踪所有的变化。主键约束将根据SQLSelect-String的执行而被创建。此值是默认值 |
| TRACKINGON | 0 | 对于接收到数据的表，SQL CE将不跟踪所有的变化。主键约束在本地表中也不会被创建 |
| TRACKINGON | 2 | 对于接收到数据的表，SQLCE将跟踪所有的变化。如果索引和主键已经存在SQL Server表中，并且字段被指定在SQLSelectString查询语句里，那么服务器表中的索引和主键将创建到CE本地表中 |
| TRACKINGON INDEXES | 3 | 对于接收到数据的表，SQLCE将不跟踪所有的变化。如果索引和主键已经存在SQL Server表中，并且字段被指定SQLSelectString查询语句里，那么服务器表中的索引和主键将创建到CE本地表中 |

需要说明的是，如果TrackOption参数被设置成TRACKINGON或TRACKINGON_INDEXES值，那么在如下情况中，调用Push方法将产生错误：

- SQL语句返回的是一个不可更新的结果集。
- 在得到的结果集中没有包含主键字段。
- SQLSelectString可以指向存储过程或视图，但存储过程或视图却引用了多个表或者引用了一个表但此表不能被更新。

参数ErrorTableName表示CE本地错误表名。在以后调用Push方法向服务器提交变化数据时，如果发生错误，此错误表将被创建，并被写入错误信息。如果调用Pull方法时，此错误表已经存在，将会发生错误。

在执行Pull方法之前，需要通过ISSCERDA接口设置InternetURL、InternetLogin、InternetPassword和LocalConnectionString属性。

- IntemetURL属性表示SQL Server CE Server Agent的URL地址。
- InternetLogin和InternetPassword属性分别为访问SQL Server CE Server Agent的用户名和密码。当然，如果SQL Server CE Server Agent的访问权限已经设置为匿名访问，只需要将这两个属性设置成空字符串即可。
- LocalConnectionString属性表示本地数据库的连接串，此连接串中指定的数据库必须存在，否则在执行Pull方法时将发生错误。

如果要了解Pull方法的执行情况，可以调用SUCCEEDED或FAILED宏来判断方法执行成功与否。

关于Pull的具体用法，将在3.6节中的远程数据访问综合示例中详细说明。

### 3.5.2 推数据

推数据（Push）和拉数据（Pull）是完全相反的两个方法。拉数据是从服务器中下载数据，推数据则是将CE本地库中的数据上传到服务器中。Push方法的定义如下：

```
virtual / * [id] * / HRESULT STDMETHODCALLTYPE Push(
/* [in] */ BSTR LocalTableName,
/* [in] */ BSTR OLEDBConnectionString,
/* [defaultvalue] [in] */ RDA_ BATCHOPTION BatchOption)= 0;
```

上述代码中，各参数的含义如下：

- 参数LocalTableName表示将向服务器上传的变化数据的表名。
- 参数OLEDBConnectionString表示连接到SQL Server服务器的连接字符串。
- 参数BatchOption表示上传数据方法。如果BatchOption的值为BATCHINGOFF，那么上传的数据将一条一条地更新到服务器上；如果BatchOption的值为BATCHINGON，那么上传的数据将一次性更新到服务器。

**注意**：BATCHINGON和BATCHINGOFF方法都会将所有的错误写入错误表，但是如果BatchOption参数被设置为BATCHINGON，此时若上传5条记录，其中3条记录发生错误，那么将没有数据被更新到服务器中，反之，如果BATCHINGOFF被设置，那么在刚才的那种情况下，还将会有2条记录被更新到服务器中。从这个例子中，就能够看出BATCHINGON和

BATCHINGOFF取值的区别了。

此外，上传数据的属性设置与下载数据的属性设置一致，这里就不重复说明了。

最后需要说明的一点就是，只有那些通过Pull方法下载并且TrackOption参数被设置成TRACKINGON或TRACKINGON_INDEXES的表，才能通过Push方法上传。

用户可以通过SUCCEEDED宏或FAILED宏来判断Push方法是否执行成功。

### 3.5.3 远程T-SQL操作

远程T-SQL操作（Submit）可以执行远程SQL语句，即可以将SQL语句传递到服务器上直接执行。Submit方法的定义如下：

```
virtual/*[id]*/HRESULT STDMETHODCALLTYPE SubmitSQL(
/*[in]*/BSTR SQLString,
/*[in]*/BSTR OLEDBConnectionString)=0;
```

上述代码中，各参数的含义如下：

- 参数SQLString表示将要执行的SQL语句。
- 参数OLEDBConnectionString为连接到SQL Server服务器的连接字符串。

在执行Submit方法前，只需要设置InternetURL、InternetLogin、InternetPassword这3个属性。关于这3个属性的含义及其设置，请参考3.5.1节。

Submit方法相对于Pull和Push方法而言，还是比较简单的，因此就不多做说明了。同样，Submit的执行情况也可以通过SUCCEEDED宏或FAILED宏来判断。

## 3.6 远程数据访问综合示例

在本节中，将利用上一节的知识来编写从SQL Server服务器获取数据、向SQL Server服务器提交数据和远程执行SQL操作等示例。下面就来逐步详细介绍。

**1. 前期准备**

本节将演示使用SQLCE实现远程数据访问的功能。在正式工作之前，需要做如下准备工作：

1）按照3.3.3节的说明进行SQLCE连接配置。

2）在PDA上安装好SQLCE 3.0，这里以DELL X51 PDA为例。

3）在服务器上建立一个dbtest数据库，并在此数据库中新建SERVERTIME和students表。其中SERVERTIME表结构如表3-13所示。

**表3-13 SERVERTIME表结构**

| 字　段 | 字段类型 | 备　注 |
|---|---|---|
| CurTime | Datetime | 表示当前时间，作为主关键字 |

在后面的例子中，将通过远程T-SQL来操作此表，例如更新CurTime字段时间等。students表的结构如表3-14所示。

表3-14 students表结构

| 字 段 | 字段类型 | 备 注 |
|---|---|---|
| Id | Int | 表示学生编号，设为主关键字 |
| Name | Varchar（20） | 表示学生姓名 |
| ClassName | Varchar（40） | 表示学生所在班级 |

应该预先在此表中存储一些数据，以方便在例子中执行Pull操作时，将服务器中students表的数据下载到PDA的数据库中。

4）在PDA上，通过isqlw30工具建立一个数据文件DB_Students.sdf，该数据文件默认放在My Documents目录下。

### 2. 新建项目

1）参考3.1.2节的介绍创建一个基于对话框的应用程序RDAExam，编译环境设置为CHSINT SDK For WinCE 6.0（ARMV4I），设备选择CHSINT SDK For WinCE 6.0 Emulator，把窗口字体改成宋体小五号。

2）在IDD_RDAEXAM_DIALOG对话框上放置3个按钮，如图3-59所示，分别用于实现SUBMIT、PULL和PUSH三大功能，将它们的标识依次设为IDC_BTN_SUBMIT、IDC_BTN_PULL、IDC_BTN_PUSH。

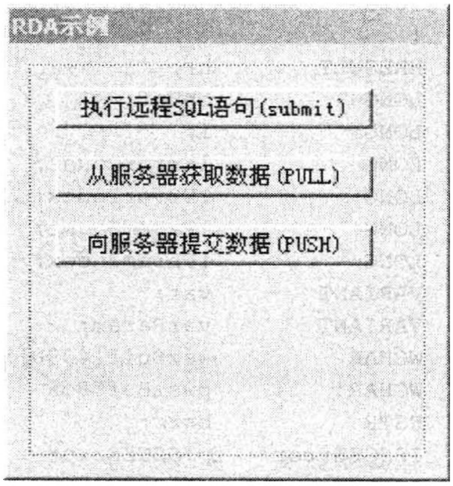



图3-59 远程数据访问综合示例界面设计

由于这里需要用到RDA定义的接口类，因此还要在RDAExamDlg.h文件中引用ca_mergex30.h文件。代码如下所示：

```
#include <ca_merge30.h>
```

### 3. 初始化COM环境

SQLCE远程数据访问是通过SQLCE提供的COM接口函数来实现相应功能。因此需要在CRDAExamDlg::OnInitDialog函数里初始化COM环境，即在OnInitDialog函数中添加如下代码：

```
//初始化COM环境
CoInitializeEx(NULL,COINIT_MULTITHREADED);
```

最后，还需要在对话框销毁事件中释放COM环境，代码如下所示：

```
//释放COM环境
CoUninitialize();
```

### 4. 实现显示执行RDA错误的方法

在执行远程数据访问的过程中，各种各样的原因都可能引发错误，SQLCE提供了ISSCEErrors接口、ISSCEError接口、ISSCEParams接口和ISSCEParam接口以获取发生错误的详

细信息。关于这四个接口的用法，在此就不详细描述了，可以参考SQLCE的帮助文档或者参考下面的ShowErrors方法。

可以使用ISSCEErrors接口、ISSCEError接口、ISSCEParams接口和ISSCEParam接口封装一个显示RDA错误方法的ShowErrors，该方法将通过弹出对话框的形式来提示错误信息。

为CRDAExamDlg类添加私有方法ShowErrors，该方法的定义如下：

```
private:
//显示远程数据访问时的错误
void  ShowErrors(ISSCEErrors*  pISSCEErrors);
```

ShowErrors方法的具体实现代码如下所示：

```
//显示远程数据访问时的错误
void    CRDAExamDlg::ShowErrors(ISSCEErrors* pISSCEErrors)
{
HRESULT        hr;
LONG           cbBuf;
LONG           i;
LONG           lErrorCount;
LONG           lErrorIndex;
LONG           lParamCount;
LONG           lParamIndex;
VARIANT        var;
VARIANT        varParam;
WCHAR          wszBuff[4096];
WCHAR*         pwszBuffPos    = &wszBuff[0];
BSTR           bstr;
ISSCEError*    pISSCEError    = NULL;
ISSCEParams*   pISSCEParams   = NULL;
ISSCEParam*    pISSCEParam    = NULL;
BOOL           fSuccess       = FALSE;

//初始化变量
VariantInit(&var);
VariantInit(&varParam);

//通过IISSCEErrors接口得到发生错误的数量
if(FAILED(hr = pISSCEErrors->get_Count(&lErrorCount)))
    goto Exit;
if(lErrorCount <= 0)
{
    ::MessageBox(NULL,L"没有错误信息",L"显示错误",MB_OK);
    fSuccess = TRUE;
    goto Exit;
}

//用弹出对话框显示发生的每一条错误
for(lErrorIndex = 0; lErrorIndex < lErrorCount; lErrorIndex++)
{
    cbBuf = swprintf(pwszBuffPos,L"ERROR  %d of %d\r\n",
        lErrorIndex+1,lErrorCount);
    pwszBuffPos += cbBuf;
```

```
//得到错误记录
var.vt = VT_I4;
var.lVal = lErrorIndex;
if(FAILED(hr = pISSCEErrors->get_Item(var, &pISSCEError)))goto Exit;

//得到错误详细信息
if(FAILED(hr = pISSCEError->get_Description(&bstr)))goto Exit;
cbBuf = swprintf(pwszBuffPos,L"DESCRIPTION: '%s'\r\n",bstr);
pwszBuffPos += cbBuf;
SysFreeString(bstr);

//错误信息编号
if(FAILED(hr = pISSCEError->get_Number(&i)))goto Exit;
cbBuf = swprintf(pwszBuffPos,L"NUMBER: %8.8X\r\n",i);
pwszBuffPos += cbBuf;

//本地错误
if(FAILED(hr = pISSCEError->get_NativeError(&i)))goto Exit;
cbBuf = swprintf(pwszBuffPos,L"NATIVE_ERROR: %d\r\n",i);
pwszBuffPos += cbBuf;

//得到错误源
if(FAILED(hr = pISSCEError->get_Source(&bstr)))goto Exit;
cbBuf = swprintf(pwszBuffPos,L"SOURCE: '%s'\r\n",bstr);
pwszBuffPos += cbBuf;
SysFreeString(bstr);

//得到错误参数信息
if(FAILED(hr = pISSCEError->get_Params(&pISSCEParams)))goto Exit;

//得到错误参数的个数
if(FAILED(hr = pISSCEParams->get_Count(&lParamCount)))goto Exit;

//显示每一个参数
for(lParamIndex = 0; lParamIndex < lParamCount; lParamIndex++)
{

    //得到错误参数对象
    var.vt = VT_I4;
    var.lVal = lParamIndex;
    if(FAILED(hr = pISSCEParams->get_Item(var, &pISSCEParam)))goto Exit;

    //得到并显示错误参数值
    if(FAILED(hr = pISSCEParam->get_Param(&varParam)))goto Exit;
    if(VT_I4 == varParam.vt || VT_UI4 == varParam.vt)
    {
        cbBuf = swprintf(pwszBuffPos,L"P%d: %d\r\n",lParamIndex,
            (LONG)varParam.lVal);
    }
```

```
        else if(VT_I2 == varParam.vt || VT_UI2 == varParam.vt)
        {
            cbBuf = swprintf(pwszBuffPos,L"P%d: %d\r\n",lParamIndex,
                (LONG)varParam.iVal);
        }
        else if(VT_BSTR == varParam.vt)
        {
            cbBuf = swprintf(pwszBuffPos,L"P%d: '%s'\r\n",lParamIndex,
                varParam.bstrVal);
            }
            pwszBuffPos += cbBuf;

            //清空参数变量
            VariantClear(&varParam);

            //释放参数对象
            pISSCEParam->Release();
            pISSCEParam = NULL;
        }
        cbBuf = swprintf(pwszBuffPos,L"\r\n");
        pwszBuffPos += cbBuf;

    }

    //用弹出框显示错误信息
    ::MessageBox(NULL,wszBuff,L"Error",MB_OK);
    fSuccess = TRUE;

Exit:
    // 释放错误参数对象
    if(pISSCEParam)
    {
        pISSCEParam->Release();
        pISSCEParam = NULL;
    }

    //释放错误参数集对象
    if(pISSCEParams)
    {
        pISSCEParams->Release();
        pISSCEParams = NULL;
    }

    //释放错误对象
    if(pISSCEError)
    {
        pISSCEError->Release();
        pISSCEError = NULL;
    }
```

```
        //如果获取错误信息失败
        if(!fSuccess)
        {
            ::MessageBox(NULL,L"得到错误信息失败!",L"显示
错误",MB_OK);
        }
        return;
    }
```

### 5. 实现SUBMIT

利用MFC类向导，新建CDlgSubmit对话框。该对话框供用户输入执行远程SQL语句的参数，窗口样式如图3-60所示。

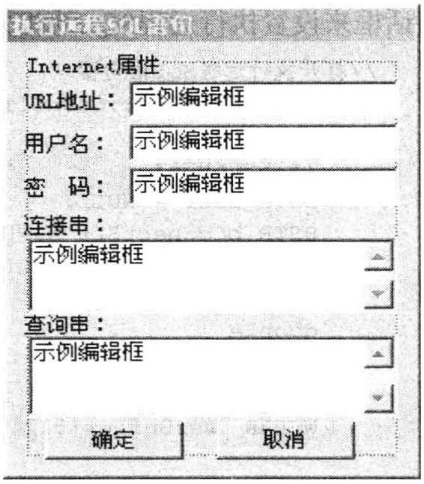


图3-60 执行远程SQL语句界面设计

对话框上的主要控件及其属性设置如表3-15所示。

**表3-15 执行远程SQL语句界面控件及其属性设置**

| ID | 描 述 |
|---|---|
| IDC_EDT_INTERNETURL | 用于输入Windows CE代理IIS的URL地址，对应成员变量为m_internetURL，类型为CString，设置其默认值为http://192.168.1.47/sqlce30/sqlcesa30.dl1 |
| IDC_EDT_INTERNETUSERNAME | 用于输入Windows CE代理IIS登录用户名，对应成员变量为m_internetUsername，型为Cstring |
| IDC_EDT_INTERNETPASSWORD | 用于输入Windows CE代理IIS登录密码，对应成员变量为m_internetPassword，类型为Cstring |
| IDC_EDT_CONNECTSTR | 用于输入连接服务器数据库的连接串，对应成员变量为m_connectStr，类型CString。选中其multiline属性，支持多行操作。设置其默认值为provider=sqloledb；data source=LJD-5；Initial Catalog=dbtest；user id=sa；password=123456 |
| IDC_ EDT_QUERYSTR | 用于要在远程服务器上执行的SQL语句，对应成员变量为m_queryStr，类型CString。选中其multiline属性，支持多行操作。设置其默认值为UPDATE_SERVERTIME_SET_CurTime=Getdate() |
| IDC_BTN_OK | 按钮控件，标题设为“确定”。在其单击事件方法里调用OnOk方法关闭对话框 |
| IDC_BTN_CANCEL | 按钮控件，标题设为“取消”。在其单击事件方法里调用OnCancel方法关闭对话框 |

参数URL地址必须输入正确，默认值是http://192.168.1.47/sqlce30/sqlcesa30.dll，这里的192.168.1.47是服务器地址，应根据自己的服务器实际地址进行设置。另外可通过PDA的IE窗口来测试此地址是否正确。

参数连接串表示访问服务器数据库的连接字符串，默认值是“provider=sqloledb； data source=LDJ-5；Initial Catalog=dbtest；user id=sa；.password= 123456”，这里wang指的是服务器网络名，应设置自己服务器的网络名，并将访问的用户名、密码设置正确。

最后，切换到IDD_ RDAEXAM_ DIALOG对话框上，在主窗体上添加“执行远程SQL语句（submit）”按钮的单击事件代码。在此单击方法里，首先通过打开IDD_DLGSUBMIT_DILAOG

对话框来设置执行远程SQL的相关属性，然后执行远程SQL语句，具体实现代码如下所示：

```
//打开执行远程SQL语句窗口
void CRDAExamDlg::OnBnClickedBtnSubmit()
{
    /*ISSCERDA*                   pCERDA = NULL;
    BSTR bStr = NULL;
    BSTR bConnectStr = NULL;
    BSTR bQueryStr = NULL;
    SYSTEMTIME                    systemtime;                //系统时间
    double                        timeTotal;                 //总共消耗的时间
    __int64                       timeStart = 0;             //开始操作时的时间
    __int64                       timeStop;                  //结束操作时的时间
    WCHAR  wszBuffer[512];


    CDlgSubmit dlgSubmit;
    if(dlgSubmit.DoModal())
    {
        //得到pCERDA接口
         HRESULT hr = CoCreateInstance(CLSID_RemoteDataAccess,NULL,CLSCTX_INPROC_
SERVER,IID_ISSCERDA, (LPVOID *)&pCERDA);
        ASSERT(hr == S_OK);
        ASSERT(pCERDA != NULL);
        //1)设置Internet URL
        bStr = SysAllocString(dlgSubmit.m_internetURL);
        pCERDA->put_InternetURL(bStr);
        SysFreeString(bStr);

        //2)设置Internet userName
        bStr = SysAllocString(dlgSubmit.m_internetUsername);
        pCERDA->put_InternetLogin(bStr);
        SysFreeString(bStr);

        //3)设置Internet password
        bStr = SysAllocString(dlgSubmit.m_internetPassword);
        pCERDA->put_InternetPassword(bStr);
        SysFreeString(bStr);

        //4)设置服务器连接串
        bConnectStr = SysAllocString(dlgSubmit.m_connectStr);
        //5)设置执行的远程SQL语句
        bQueryStr = SysAllocString(dlgSubmit.m_queryStr);

        //记录执行SQL语句前的时间
        GetLocalTime(&systemtime);
        SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStart);

        //执行远程SQL语句
        hr = pCERDA->SubmitSQL(bQueryStr,bConnectStr);

        SysFreeString(bQueryStr);
        SysFreeString(bConnectStr);
```

```
            if(SUCCEEDED(hr))
            {
            //记录执行SQL语句后的时间
                GetLocalTime(&systemtime);
                SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStop);
                //记录执行SQL语句总共消耗的时间
                timeTotal = ((double)(timeStop - timeStart) / 10000000);
                swprintf(wszBuffer,_T("执行远程SQL语句成功\r\n\r\总共消耗= %8.3f
秒"),timeTotal); */
                AfxMessageBox(_T("执行远程SQL语句成功\r\n\r\总共消耗= 2.000秒"));
                /*}AfxMessageBox(wszBuffer);
            else
            {
                //显示执行远程SQL语句错误
                ISSCEErrors  *pISSCEErrors = NULL;
                if(SUCCEEDED(hr = pCERDA->get_ErrorRecords(&pISSCEErrors)))
                {
                    ShowErrors(pISSCEErrors);
                    pISSCEErrors->Release();
                }
            }
            //释放远程访问接口
            pCERDA->Release();
        }*/
    }
```

由于这里引用了CDlgSubmit类，因此要在RDAExamDlg.cpp文件中引用DlgSubmit.h。到此，执行远程SQL语句的功能就实现了，执行效果如图3-61所示。当然，在执行远程SQL时，要确保服务器URL地址、服务器数据库连接串正确；还要确保执行的SQL语句的合法性。

### 6. 实现PULL

利用MFC类向导，新建CDlgPull对话框。此对话框主要供用户输入执行Pull时的参数，对话框样式如图3-62所示。

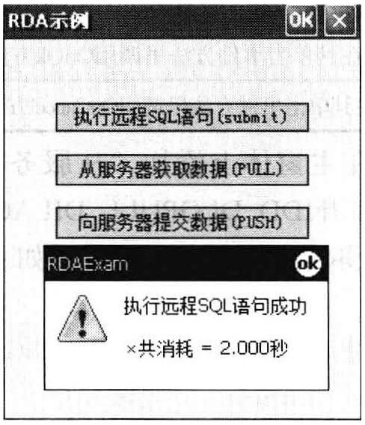



图3-61　执行远程SQL语句示例效果图

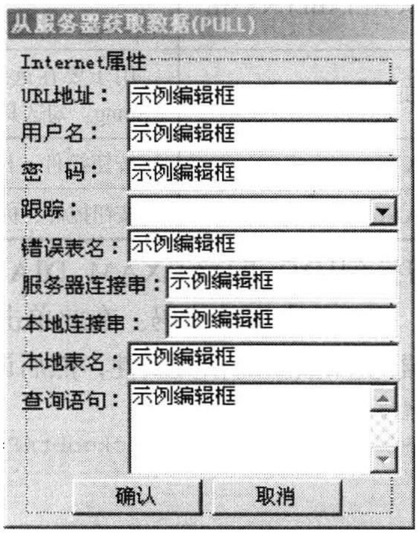



图3-62　从服务器获取数据（PULL）界面设计

对话框上的主要控件及其属性设置如表3-16所示。

表3-16 从服务器获取数据（PULL）界面控件及其属性设置

| ID | 描 述 |
|---|---|
| IDC_EDT_INTERNETURL | 用于输入Windows CE代理IIS的URL地址，对应成员变量为m_internetURL，类型为Cstring，设置其默认值为http:// 192.168.1.47/sqlce30/sqlcesa30.dll |
| IDC_EDT_<br>INTERNETUSERNAME | 用于输入Windows CE代理IIS登录用户名，对应成员变量为m_internetUsername，类型为CString |
| IDC_EDT_<br>INTERNETPASSWORD | 用于输入Windows CE代理IIS登录密码，对应成员变量为m_ internetPa_ssword，类型为CString |
| IDC_CMB_TRACKING | 组合框，用于设置获取数据时的跟踪类型，对应成员变量为m_tracking，类型为CString。同时将此组合框设置成DROP_LIST，并且将下拉框的内容设为：<br>TRACKINGON<br>TRACKINGOFF<br>TRACKINGON_INDEXES<br>TRACKINGOFF_INDEXES<br>最后去掉排序（sort）属性 |
| IDC_EDT_ERRORTABLENAME | 用于输入错误表名称，对应成员变量m_errorTableName类型为Cstring，设置其默认值为tblErrors |
| IDC_EDT_CONNECTSTR | 用于输入连接服务器数据库的连接串，对应成员变量为m_connectStr，类型CString。勾选其multiline属性，以支持多行操作。设置其默认值为provider=sqloledb; data source=LDJ-5; InitialCatalog=dbtest; user id=sa; password=123456 |
| IDC_EDT_LOCALCONNECTSTR | 用于输入连接本地数据库的连接字符串，对应成员变量m_localConnectStr，类型为Cstring，设置其默认值为Provider=microsoft.sqlserver.oledb.ce.3.0; DataSource=\\My Documents\\DB_Students.sdf |
| IDC_EDT_LOCALTABLENAME | 用于输入存储数据的本地表名，对应成员变量m_localTableName，类型为CString，设置其默认值为students |
| IDC_EDT_QUERYSTR | 用于要在远程服务器上执行的查询语句，对应成员变量为m_queryStr，类型CString。勾选其multiline属性，以支持多行操作。设置其默认值为select * from students |
| IDC_BTN_OK | 按钮控件，标题设为“确定”。在其单击事件方法里调用OnOk方法关闭对话框 |
| IDC_BTN_CANCEL | 按钮控件，标题设为“取消”。在其单击事件方法里调用OnCancel方法关闭对话框 |

最后切换到IDD_RDAEXAM_DIALOG对话框，在主窗体上添加“从服务器获取数据（PULL）”按钮单击事件代码。在此单击方法里，首先打开IDD_DLGPULL_DILAOG对话框，设置从服务器获取数据的相关属性，然后调用Pull方法来获取数据，具体实现代码如下所示：

```
//从远程服务器获取数据
void CRDAExamDlg::OnBnClickedBtnPull()
{
    //
    /*ISSCERDA*           pCERDA = NULL;
```

```
	BSTR bStr = NULL;
	BSTR bConnectStr = NULL;
	BSTR bQueryStr = NULL;
	BSTR bLocalConnectStr = NULL;
	BSTR bLocalTableName = NULL;
	BSTR bErrorTableName  = NULL;
	RDA_TRACKOPTION trackOption = TRACKINGOFF;

	SYSTEMTIME systemtime;                          //系统时间
	double timeTotal;                               //总共消耗的时间
	__int64 timeStart = 0;                          //开始操作时的时间
	__int64 timeStop;                               //结束操作时的时间
	WCHAR wszBuffer[512];

	CDlgPull dlgPull;
	if(dlgPull.DoModal())
	{
		//得到pCERDA接口
		HRESULT hr = CoCreateInstance(CLSID_RemoteDataAccess,NULL,CLSCTX_
INPROC_SERVER,IID_ISSCERDA, (LPVOID *)&pCERDA);
		ASSERT(hr == S_OK);
		ASSERT(pCERDA != NULL);
		//1)设置Internet URL
		bStr = SysAllocString(dlgPull.m_internetURL);
		pCERDA->put_InternetURL(bStr);
		SysFreeString(bStr);

		//2)设置Internet userName
		bStr = SysAllocString(dlgPull.m_internetUsername);
		pCERDA->put_InternetLogin(bStr);
		SysFreeString(bStr);

		//3)设置Internet password
		bStr = SysAllocString(dlgPull.m_internetPassword);
		pCERDA->put_InternetPassword(bStr);
		SysFreeString(bStr);

		//4)得到跟踪类型
		if(dlgPull.m_tracking == L"TRACKINGON")
		{
			trackOption = TRACKINGON;
		}else if(dlgPull.m_tracking == L"TRACKINGOFF")
		{
			trackOption = TRACKINGOFF;
		}else if(dlgPull.m_tracking == L"TRACKINGON_INDEXES")
		{
			trackOption = TRACKINGON_INDEXES;
		}else if(dlgPull.m_tracking == L"TRACKINGOFF_INDEXES")
		{
			trackOption = TRACKINGOFF_INDEXES;
		}else
		{
```

```
        trackOption = TRACKINGOFF;
}

//5)设置错误表名
bErrorTableName = SysAllocString(dlgPull.m_errorTableName);

//6)设置服务器连接串
bConnectStr = SysAllocString(dlgPull.m_connectStr);

//7)设置本地数据库连接串
bLocalConnectStr = SysAllocString(dlgPull.m_localConnectStr);
pCERDA->put_LocalConnectionString(bLocalConnectStr);
SysFreeString(bStr);

//8)设置本地表名
bLocalTableName = SysAllocString(dlgPull.m_localTableName);

//9)设置查询的SQL语句
bQueryStr = SysAllocString(dlgPull.m_queryStr);

//记录获取服务器数据前的时间
GetLocalTime(&systemtime);
SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStart);

//获取服务器数据
hr = pCERDA->Pull(bLocalTableName,
                  bQueryStr,
                  bConnectStr,
                  trackOption,
                  bErrorTableName);

SysFreeString(bLocalTableName);
SysFreeString(bQueryStr);
SysFreeString(bConnectStr);
SysFreeString(bErrorTableName);

if(SUCCEEDED(hr))
{
        //记录获取服务器数据后的时间
        GetLocalTime(&systemtime);
        SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStop);
  //记录获取服务器数据的总时间
        timeTotal = ((double)(timeStop - timeStart) / 10000000);
        swprintf(wszBuffer,_T("从服务器获取数据成功\r\n\r\总共消耗= %8.3f
秒"),timeTotal);
        */AfxMessageBox(_T("从服务器获取数据成功\r\n\r\总共消耗= 5.000秒"));
/*}
else
{
        //显示获取服务器数据错误
        ISSCEErrors  *pISSCEErrors = NULL;
        if(SUCCEEDED(hr = pCERDA->get_ErrorRecords(&pISSCEErrors)))
        {
                ShowErrors(pISSCEErrors);
```

```
                    pISSCEErrors->Release();
                }
            }
            //释放远程访问接口
            pCERDA->Release();
    }*/
}
```

由于这里引用了CDlgPull类，因此还要在RDAExamDlg.cpp文件中引用DlgPull.h。至此，从服务器获取数据的示例就编写完成了，其运行效果如图3-63所示。

**注意：**从远程服务器获取数据时，要确保本地数据库存在，并且指定的本地数据表不存在。

### 7. 实现PUSH

利用MFC类向导，新建CDlgPush对话框。此对话框主要供用户输入执行Push时的参数，对话框样式如图3-64所示。

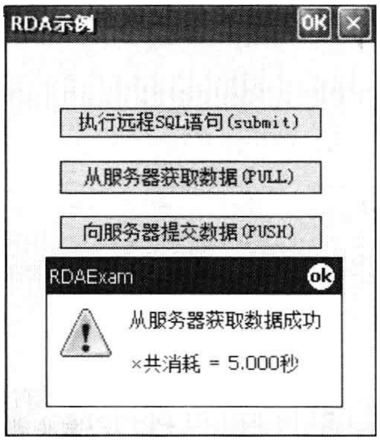



图3-63 从服务器获取数据（PULL）示例效果图

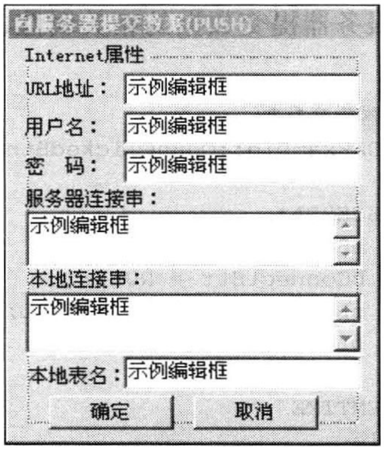



图3-64 向服务器提交数据（PUSH）界面设计

对话框上的主要控件及其属性设置如表3-17所示。

**表3-17 向服务器提交数据（PUSH）界面控件及其属性设置**

| ID | 描 述 |
|---|---|
| IDC_EDT_INTERNETURL | 用于输入Windows CE代理IIS的URL地址，对应成员m_internetURL，类型为CString，设置其默认值为http://192.168.1.47/sqlce30/sqlcesa30.dll |
| IDC_EDT_INTERNETUSERNAME | 用于输入Windows CE代理IIS登录用户名，对应m_internetUsername，类型为CString |
| IDC_EDT_INTERNETPASSWORD | 用于输入Windows CE代理IIS登录密码，对应成员变量为m_internetPassword，类型为CString |
| IDC_EDT_CONNECTSTR | 用于输入连接服务器数据库的连接串，对应成员变量为m_connectStr，类型CString。勾选其multiline属性，以支持多行操作。设置其默认值为provider=sqloledb；data source=LDJ-5；Initial Catalog=dbtest；user id=sa；password=123456 |

（续）

| ID | 描　述 |
|---|---|
| IDC_EDT_LOCALCONNECTSTR | 用于输入连接本地数据库的连接字符串，对应成员变量m_localConnectStr，类型为CString。设置其默认值为Provider=　microsoft.sqlserver.oledb.ce.3.0；Data Source=\\My Documents\\ DB_Students.sdf |
| IDC_EDT_LOCALTABLENAME | 用于输入存储数据的本地表名，对应成员变量m_localTableName，类型为Cstring。设置其默认值为students |
| IDC_BTN_OK | 按钮控件，标题设为“确定”。在其单击事件方法里调用OnOk方法关闭对话框 |
| IDC_BTN_CANCEL | 按钮控件，标题设为“取消”。在其单击事件方法里调用OnCancel方法关闭对话框 |

最后切换到IDD_RDAEXAM_DIALOG对话框，在主窗体上添加“向服务器提交数据（PUSH）”按钮单击事件代码。在此单击方法里，首先打开IDD_DLGPUSH_DILAOG对话框，设置向服务器提交数据的相关属性，然后通过调用Push方法来获取数据，具体实现代码如下所示：

```
//向服务器提交数据
void CRDAExamDlg::OnBnClickedBtnPush()
{
    /*ISSCERDA*                     pCERDA = NULL;
    BSTR bStr = NULL;
    BSTR bConnectStr = NULL;
    BSTR bLocalConnectStr = NULL;
    BSTR bLocalTableName = NULL;

    SYSTEMTIME                    systemtime;                  //系统时间
    double                        timeTotal;                   //总共消耗的时间
    __int64                       timeStart = 0;               //开始操作时的时间
    __int64                       timeStop;                    //结束操作时的时间
    WCHAR   wszBuffer[512];

    CDlgPush dlgPush;
    if(dlgPush.DoModal())
    {
           //得到pCERDA接口
           HRESULT  hr  =  CoCreateInstance(CLSID_RemoteDataAccess,NULL,CLSCTX_
INPROC_SERVER,IID_ISSCERDA,  (LPVOID *)&pCERDA);
           ASSERT(hr == S_OK);
           ASSERT(pCERDA != NULL);
           //1)设置Internet URL
           bStr = SysAllocString(dlgPush.m_internetURL);
           pCERDA->put_InternetURL(bStr);
           SysFreeString(bStr);

           //2)设置Internet userName
           bStr = SysAllocString(dlgPush.m_internetUsername);
           pCERDA->put_InternetLogin(bStr);
           SysFreeString(bStr);
```

```
            //3)设置Internet password
            bStr = SysAllocString(dlgPush.m_internetPassword);
            pCERDA->put_InternetPassword(bStr);
            SysFreeString(bStr);


            //4)设置服务器连接串
            bConnectStr = SysAllocString(dlgPush.m_connectStr);

            //5)设置本地数据库连接串
            bLocalConnectStr = SysAllocString(dlgPush.m_localConnectStr);
            pCERDA->put_LocalConnectionString(bLocalConnectStr);
            SysFreeString(bStr);

            //6)设置本地表名
            bLocalTableName = SysAllocString(dlgPush.m_localTableName);


            //记录向服务器提交数据前的时间
            GetLocalTime(&systemtime);
            SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStart);

        //向服务器提交数据
            hr = pCERDA->Push(bLocalTableName,bConnectStr,BATCHINGOFF);

            SysFreeString(bLocalTableName);
            SysFreeString(bConnectStr);

            if(SUCCEEDED(hr))
            {
                    //记录向服务器提交数据后的时间
                    GetLocalTime(&systemtime);
                    SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStop);
                    //记录向服务器提交数据的总时间
                    timeTotal = ((double)(timeStop - timeStart) / 10000000);
                    swprintf(wszBuffer,_T("向服务器提交数据成功\r\n\r\总共消耗= %8.3f
秒"),timeTotal);
                    */AfxMessageBox(_T("向服务器提交数据成功\r\n\r\总共消耗= 1.000秒"));
            /*}
            else
            {
                    //显示向服务器提交数据失败信息
                    ISSCEErrors  *pISSCEErrors = NULL;
                    if(SUCCEEDED(hr = pCERDA->get_ErrorRecords(&pISSCEErrors)))
                    {
                            ShowErrors(pISSCEErrors);
                            pISSCEErrors->Release();
                    }
            }
            //释放远程访问接口
            pCERDA->Release();
    }*/
}
```

由于这里引用了CDlgPush类，因此还需要在RDAExamDlg.cpp文件中引用DlgPush.h。至此，向服务器提交数据的示例就编写完成了，其运行效果如图3-65所示。

**注意**：当向服务器提交表，在从服务器获取数据时，应将其跟踪属性设置成TRACKINGON或者TRACKINGON_INDEXES。

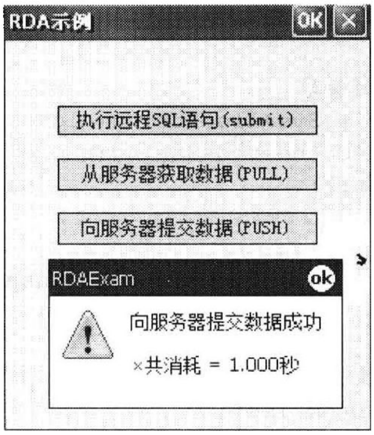



图3-65 向服务器提交数据示例运行效果

# 第二部分　硬件篇

- ◆ 第4章　S3C6410硬件体系结构
- ◆ 第5章　Windows CE Boot Loader与OAL
- ◆ 第6章　接口驱动程序开发

# 第4章　S3C6410硬件体系结构

## 4.1　S3C6410微处理器

S3C6410是一款SAMSUNG公司基于ARM1176的16/32位的高性能、低功耗、高性价比的RSIC通用微处理器，适用于手持、移动等终端设备。其为2.5G和3G通信服务提供了优化的硬件性能。S3C6410采用64/32bit的内部总线架构，融合了AXI、AHB、APB总线。另外其还有很多强大的硬件加速器，可用于运动视频处理、音频处理、2D加速、显示处理和缩放。为满足高端通信服务中的数据带宽要求，S3C6410设计了优化的外部存储器接口。为降低整个系统的成本和提升总体功能，其还设计了很多硬件功能外设。

### 4.1.1　S3C6410结构图

S3C6410结构图如图4-1所示。

### 4.1.2　S3C6410处理器特性

#### 1. ARM1176JZF-S处理器

S3C6410采用ARM1176JZF-S处理器。ARM1176JZF-S处理器包含16KB的指令数据Cache和16KB的指令数据TCM。ARM Core电压为1.1V的时候；可以运行到553MHz；在1.2V的情况下，可以运行到667MHz。通过AXI、AHB和APB组成的64/32bit内部总线和外部模块相连。支持矢量浮点型（VFP）协处理器。

#### 2. 内存子系统

- DRAM控制器：两个片选，支持SDRAM、DDR SDRAM、mobile SDRAM和mobile DDR SDRAM。每个片选最大支持256MB。
- NandFlash控制器：支持SLC/MLC NandFlash，支持512/2048Bytes Page的Nandflash，支持8Bit Nandflash，支持1/4/8Bit ECC校验，支持NandFlash Boot功能。
- OneNAND控制器：支持2个OneNAND控制器，可外接16Bit OneNand Flash，支持同步异步读取数据，支持OneNAND Boot功能。
- SROM控制器：6个片选，支持SRAM、ROM和NOR Flash，支持8/16Bit，每个片选支持128MB。

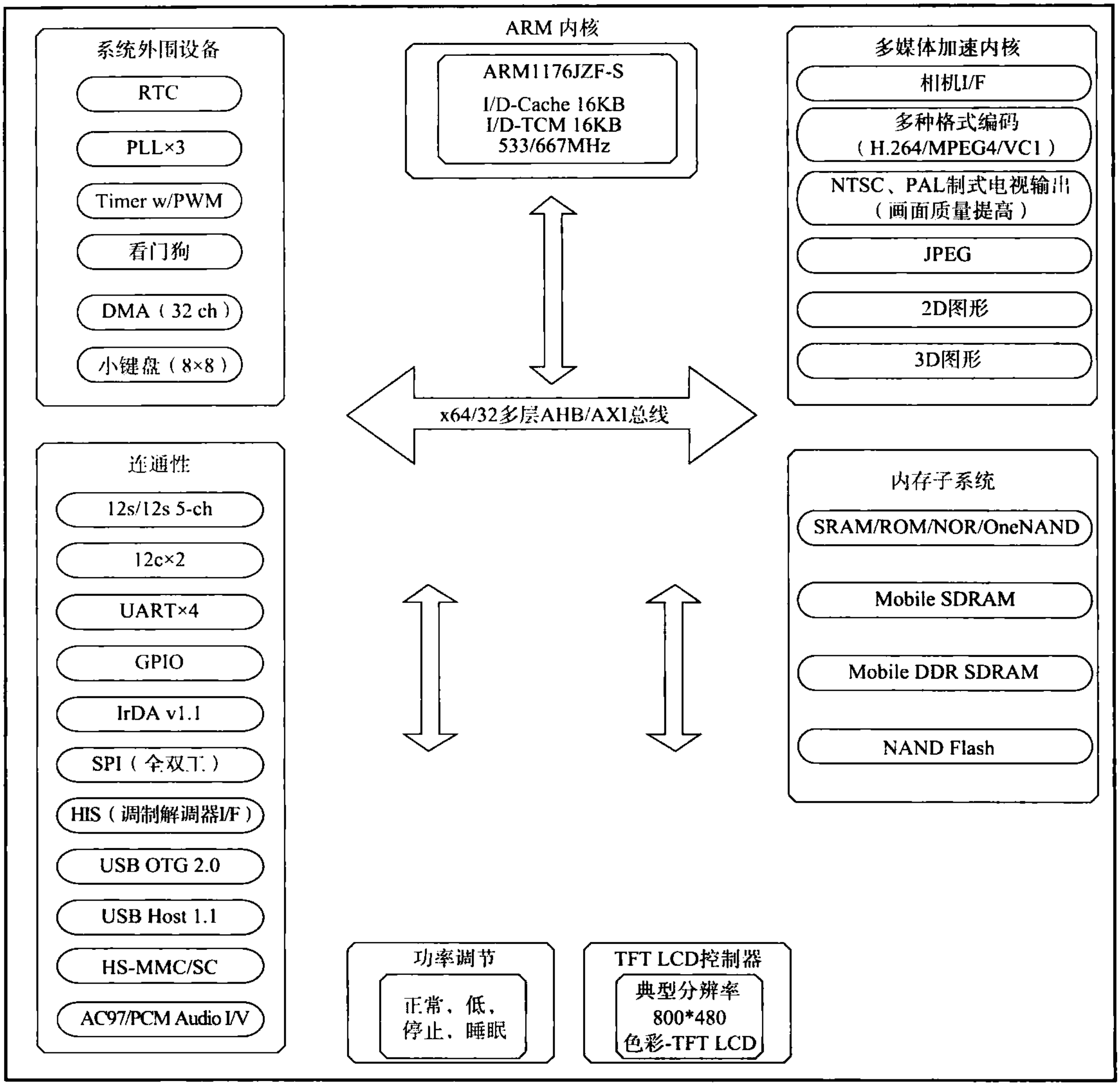



图4-1 S3C6410结构图

### 3. 显示控制器

S3C6410的显示控制器支持TFT 24Bit LCD屏，分辨率能支持到1024×1024。显示输出接口支持RGB接口、I80接口、BT.601输出（YUV422 8Bit）和输出给TV Encoder的接口。支持最多5个图形窗口并可进行Overlay操作，从Window 0到Window 4，分别支持不同的图像输入源和不同的图像格式。实际上，显示控制器可以接收来自Carema、Frame Buffer和其他模块的图像数据，可以对这些不同的图像进行Overlay，并输出到不同的接口，比如LCD屏、TV Encoder。

### 4. 系统外设

- 系统时钟：系统掉电的时候由备份电池供电，需外接32.768kHz时钟，年/月/日/时/分/秒都是BCD码格式。
- PLL时钟：支持3个PLL、分别是APLL、MPLL和EPLL。APLL为ARM提供时钟，产生ARMCLK。MPLL为所有和AXI/AHB/APB相连的模块提供时钟，产生HCLK和PCLK。EPLL为特殊的外设提供时钟，产生SCLK。
- 定时器：支持5个32Bit定时器，其中Timer0和Timer1具有PWM功能，而Timer2、Timer3和Timer4没有输出引脚，为内部定时器。
- 看门狗：也可以当做16Bit的内部定时器。
- DMA控制器：支持4个DMA控制器，每个控制器包含8个通道，支持8/16/32Bit传输，支持优先级，通道0优先级最高。
- 键盘接口：支持8×8键盘，与GPIO复用，按下和抬起按键都可产生中断。
- A/D转换和触摸屏接口：支持8个通道复用ADC转换器，10位分辨率。

### 5. 外围设备接口

- I2S总线立体声接口：用于与外接的音频编解码器通信。支持普通的I2S双通道，也支持5.1通道I2S传输，音频数据可以是8/16/32Bit，采样率从8kHz到192kHz。
- I2C总线接口：支持2个I2C控制器，双向数据传输可在高达100 kbps的标准模式下操作，在快速模式高达400 kbps。
- UART串行接口：支持4个UART口，支持DMA和中断模式，UART0/1/2还支持IrDA1.0功能。UART最高速度达3Mbps。
- GPIO输入输出：通用GPIO端口，功能复用。
- IrDA：独立的IrDA控制器，兼容IrDA1.1，支持MIR和FIR模式。
- SPI总线接口：支持2个通道串行外设接口，支持DMA和中断模式，全双工发送/接收速率达50Mbps。
- Modem调制解调器接口：内置8KB SRAM用于S3C6410和外接Modem交换数据，该SRAM还可以为Modem提供Boot功能。
- USB OTG高速接口：支持USB OTG 2.0，同时支持Slave和Host功能，最高速度达480Mbps。
- USB主设备接口：两个端口USB主设备控制器，支持USB Host1.1，全速高达12 Mbps。
- SD/MMC控制器：兼容SD Host 2.0、SD Memory Card 2.0、SDIO Card 1.0和High-Speed MMC。
- PCM串行音频接口：支持两个PCM串行音频接口，传输单声道16Bit音频数据。
- AC97音频编解码器接口：支持独立的PCM立体声音频输入、单声道MIC输入和PCM立体声音频输出，通过AC-Link接口与音频编解码器相连。

### 6. 多媒体加速特性

- 照相机接口：支持ITU-R BT.601/656 8bit标准输入。支持Zoom In功能，最大图像达

4096×4096。支持Preview，在Preview时支持Rotation和Mirror功能，Preview输出图像格式可以是RGB 16/18/24Bit和YUV420/433格式，支持图像的一些特效。

- 多格式编解码器：支持MPEG4 Simple Profile、H.264/AVC Baseline Profile、H.263 P3和VC-1 Main Profile编解码功能。支持1/2和1/4像素的运动估计、MPEG-4 AC/DC预测、H.264/AVC帧内预测；对于MPEG-4还支持可逆VLC和Data Partition功能；支持码流控制（CBR或者VBR）；编解码同时进行的时候，可支持VGA 30fps。
- 视频编码器：支持将数字视频转换成模拟的复合视频，支持N制和P制，支持Contrast、Brightness、Gamma等控制，支持复合视频和S端子输出。输入视频数据可以来自TV Scaler模块，该模块可以对视频数据进行处理。支持Resize功能，支持RGB与YUV两个不同色彩空间的转换。输入TV Scaler模块的图像最大可以是800×2048，输出图像最大是2048×2048。输出数据给TV Encoder进行编码，然后输出模拟视频。
- 翻转模块：支持对YUV420/422和RGB565/888的数据进行硬件翻转。
- 图像处理模块：类似TV Scaler模块。输入图像最大为4096×4096，输出图像最大为2048×2048，支持RGB与YUV之间的转换。
- PEG编解码器：支持JPEG编解码功能，最大尺寸为4096×4096。
- 2D加速：支持画点/线、Bitblt功能和Color Expansion。
- 3D加速。

7. 电源管理

目前支持Normal、Idle、Stop和Sleep模式。Normal是正常模式，其他模式都为不同程度的低功耗模式。Sleep模式是最低功耗模式，可以被有限的中断唤醒。

# 4.2 GPIO输入输出接口

## 4.2.1 GPIO输入输出接口简介

GPIO（General Purpose I/O，通用输入/输出接口）也称为并行I/O（parallel I/O），是最基本的I/O形式，由一组输入引脚、输出引脚或输入/输出引脚组成，CPU对它们能够进行读写操作。GPIO引脚能够通过软件编程改变输入/输出方向。

S3C6410提供了GPA～GPQ共17个GPIO组，共有187个多功能复用输入／输出端口（I/O口），每个引脚的功能必须在启动主程序之前进行定义。大多数的引脚端都是复用的，需要定义每一个引脚端功能。每个引脚端的功能通过端口控制寄存器来定义（配置）。与配置I/O口相关的寄存器包括：端口控制寄存器、端口数据寄存器、端口上拉寄存器和外部中断控制寄存器等。图4-2所示为GPIO的模块图。

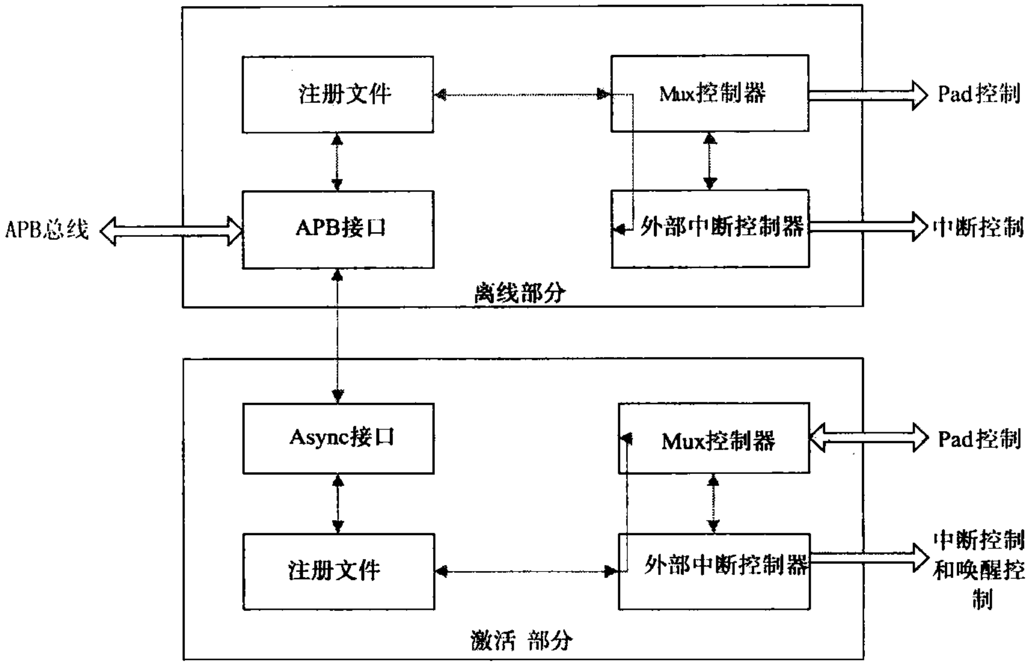



图4-2 GPIO模块图

## 4.2.2 GPIO专用寄存器

（1）端口控制寄存器（GPACON-GPQCON）

在S3C6410中大部分端口都是复用的，因此需要决定每个引脚使用哪个功能。端口控制寄存器GPnCON决定每个引脚的功能。如GPA0–GPA7与UART RXD[0]、TXD[0]、CTSn[0]、RTSn[0]、UART RXD[1]、TXD[1]、CTSn[1]、RTSn [1]等串口信号复用 。

（2）端口数据寄存器（GPADAT-GPQDAT）

如果端口被配置成输出端口，可以向GPnDAT中的相关位写入数据；如果端口被配置成输入端口，可以从GPnDAT中的相关位读入数据。

（3）端口上拉电阻寄存器（GPAUP-GPQUP）

端口上拉电阻寄存器控制每个端口组的上拉电阻的使能和禁止。当相关位为0，上拉电阻使能；当相关位为1，上拉电阻禁止。

### 1. 端口A控制寄存器

端口A控制寄存器包括5个控制寄存器，如表4-1所示，分别是GPACON、GPADAT、GPAPUD、GPACONSLP和GPAPUDSLP。

**表4-1 端口A控制寄存器的5个控制寄存器**

| 寄存器 | 地 址 | 读/写 | 描 述 | 复位值 |
| --- | --- | --- | --- | --- |
| GPACON | 0x7F008000 | 读/写 | 端口A 配置寄存器 | 0x0000 |
| GPADAT | 0x7F008004 | 读/写 | 端口A 数据寄存器 | 未定义 |

（续）

| 寄存器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| GPAPUD | 0x7F008008 | 读/写 | 端口A上拉寄存器 | 0x0005555 |
| GPACONSLP | 0x7F00800C | 读/写 | 端口A 睡眠模式配置寄存器 | 0x0 |
| GPAPUDSLP | 0x7F008010 | 读/写 | 端口A睡眠模式上拉/下拉寄存器 | 0x0 |

（1）端口A控制寄存器

端口A控制寄存器的位分配如表4-2所示。

表4-2　端口A控制寄存器的位分配

| GPACON | 位 | 描　述 | | 初始状态 |
|---|---|---|---|---|
| GPA0 | [3:0] | 0000=输入<br>0010=UART RXD[0]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[0] | 0000 |
| GPA1 | [7:4] | 0000=输入<br>0010=UART TXD[0]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[1] | 0000 |
| GPA2 | [11:8] | 0000=输入<br>0010=UART CTSn[0]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[2] | 0000 |
| GPA3 | [15:12] | 0000=输入<br>0010=UART RTSn [0]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[3] | 0000 |
| GPA4 | [19:16] | 0000=输入<br>0010=UART RXD[1]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[4] | 0000 |
| GPA5 | [23:20] | 0000=输入<br>0010=UART RTXD[1]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[5] | 0000 |
| GPA6 | [27:24] | 0000=输入<br>0010=UART CTSn [ 1]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[6] | 0000 |
| GPA7 | [31:28] | 0000=输入<br>0010=UART RTSn [1]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011=保留<br>0101=保留<br>0111=外部中断组1[7] | 0000 |

（2）端口A数据寄存器

端口A数据寄存器的位分配如表4-3所示。

**表4-3 端口A数据寄存器的位分配**

| GPADAT | 位 | 描 述 |
|---|---|---|
| GPA[7:0] | [7:0] | 当端口作为输入端口时，相应的位管脚状态，当端口作为输出端口时，管脚状态等同于相应的位。当端口作为功能管脚时，读取未被定义的值 |

（3）端口A上拉寄存器

端口A上拉寄存器的位分配如表4-4所示。

**表4-4 端口A上拉寄存器的位分配**

| GPAPUD | 位 | 描 述 |
|---|---|---|
| GPA[n] | [2n+1:2n]<br>n=0～7 | 00=禁止上拉/下拉<br>01=下拉使能<br>10=上拉使能<br>11=保留 |

（4）端口A睡眠模式配置寄存器

端口A睡眠模式配置寄存器的位分配如表4-5所示。

**表4-5 端口A睡眠模式配置寄存器的位分配**

| GPACONSLP | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| GPA[n] | [2n+1:2n]<br>n=0～7 | 00=输出0<br>01=输出1<br>10=输入<br>11=与先前状态相同 | 00 |

（5）端口A睡眠模式上拉/下拉寄存器

端口A睡眠模式上拉/下拉寄存器的位分配如表4-6所示。

**表4-6 端口A睡眠模式上拉/下拉寄存器的位分配**

| GPAPUDSLP | 位 | 描 述 |
|---|---|---|
| GPA[n] | [2n+1:2n]<br>n=0～7 | 00=禁止上拉/下拉<br>01=下拉使能<br>10=上拉使能<br>11=保留 |

### 2. 端口B控制寄存器

端口B控制寄存器包括5个控制寄存器，如表4-7所示，分别是GPBCON、GPBDAT、GPBPUD、GPBCONSLP和GPBPUDSLP。

表4-7 端口B控制寄存器的5个控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPBCON | 0x7F008020 | 读/写 | 端口B 配置寄存器 | 0x40000 |
| GPBDAT | 0x7F008024 | 读/写 | 端口B 数据寄存器 | 未定义 |
| GPBPUD | 0x7F008028 | 读/写 | 端口B上拉寄存器 | 0x00005555 |
| GPBCONSLP | 0x7F00802C | 读/写 | 端口B 睡眠模式配置寄存器 | 0x0 |
| GPBPUDSLP | 0x7F008030 | 读/写 | 端口B睡眠模式上拉/下拉寄存器 | 0x0 |

### 3. 端口C控制寄存器

端口C控制寄存器包括5个控制寄存器，如表4-8所示：分别是GPCCON、 GPCDAT、GPCPUD、GPCCONSLP和GPCPUDSLP。

表4-8 端口C控制寄存器的5个控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPCCON | 0x7F008040 | 读/写 | 端口C 配置寄存器 | 0x0000 |
| GPCDAT | 0x7F008044 | 读/写 | 端口C 数据寄存器 | 未定义 |
| GPCPUD | 0x7F008048 | 读/写 | 端口C上拉寄存器 | 0x00005555 |
| GPCCONSLP | 0x7F00804C | 读/写 | 端口C 睡眠模式配置寄存器 | 0x0 |
| GPCPUDSLP | 0x7F008050 | 读/写 | 端口C睡眠模式上拉/下拉寄存器 | 0x0 |

### 4. 端口D控制寄存器

端口D控制寄存器包括5个控制寄存器，如表4-9所示，分别是GPDCON、 GPDDAT、GPDPUD、GPDCONSLP和GPDPUDSLP。

表4-9 端口D控制寄存器的5个控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPDCON | 0x7F008060 | 读/写 | 端口D 配置寄存器 | 0x00 |
| GPDDAT | 0x7F008064 | 读/写 | 端口D数据寄存器 | 未定义 |
| GPDPUD | 0x7F008068 | 读/写 | 端口D上拉寄存器 | 0x00000155 |
| GPDCONSLP | 0x7F00806C | 读/写 | 端口D睡眠模式配置寄存器 | 0x0 |
| GPDPUDSLP | 0x7F008070 | 读/写 | 端口D睡眠模式上拉/下拉寄存器 | 0x0 |

### 5. 端口E控制寄存器

端口E控制寄存器包括5个控制寄存器，如表4-10所示，分别是GPECON、GPEDAT、

GPEPUD、GPECONSLP和GPEPUDSLP。

表4-10 端口E控制寄存器的5个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPECON | 0x7F008080 | 读/写 | 端口E配置寄存器 | 0x00 |
| GPEDAT | 0x7F008084 | 读/写 | 端口E数据寄存器 | 未定义 |
| GPEPUD | 0x7F008088 | 读/写 | 端口E上拉寄存器 | 0x00000155 |
| GPECONSLP | 0x7F00808C | 读/写 | 端口E睡眠模式配置寄存器 | 0x0 |
| GPEPUDSLP | 0x7F008090 | 读/写 | 端口E睡眠模式上拉/下拉寄存器 | 0x0 |

### 6. 端口F控制寄存器

端口F控制寄存器包括5个控制寄存器，如表4-11所示，分别是GPFCON、GPFDAT、GPFPUD、GPFCONSLP和GPFPUDSLP。

表4-11 端口F控制寄存器的5个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPFCON | 0x7F0080A0 | 读/写 | 端口F 配置寄存器 | 0x00 |
| GPFDAT | 0x7F0080A4 | 读/写 | 端口F数据寄存器 | 未定义 |
| GPFPUD | 0x7F0080A8 | 读/写 | 端口F上拉寄存器 | 0x55555555 |
| GPFCONSLP | 0x7F0080AC | 读/写 | 端口F睡眠模式配置寄存器 | 0x0 |
| GPFPUDSLP | 0x7F0080B0 | 读/写 | 端口F睡眠模式上拉/下拉寄存器 | 0x0 |

### 7. 端口G控制寄存器

端口G控制寄存器包括5个控制寄存器，如表4-12所示，分别是GPGCON、GPGDAT、GPGPUD、GPGCONSLP和GPGPUDSLP。

表4-12 端口G控制寄存器的5个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPGCON | 0x7F0080C0 | 读/写 | 端口G 配置寄存器 | 0x00 |
| GPGDAT | 0x7F0080C4 | 读/写 | 端口G数据寄存器 | 未定义 |
| GPGPUD | 0x7F0080C8 | 读/写 | 端口G上拉寄存器 | 0x00001555 |
| GPGCONSLP | 0x7F0080CC | 读/写 | 端口G 睡眠模式配置寄存器 | 0x0 |
| GPGPUDSLP | 0x7F0080D0 | 读/写 | 端口G睡眠模式上拉/下拉寄存器 | 0x0 |

### 8. 端口H控制寄存器

端口H控制寄存器包括6个控制寄存器，如表4-13所示，分别是GPHCON0、GPHCON1、GPHDAT、GPHPUD、GPHCONSLP和GPHPUDSLP。

表4-13 端口H控制寄存器的6个控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPHCON0 | 0x7F0080E0 | 读/写 | 端口H 配置寄存器0 | 0x00 |
| GPHCON1 | 0x7F0080E4 | 读/写 | 端口H 配置寄存器1 | 0x00 |
| GPHDAT | 0x7F0080E8 | 读/写 | 端口H数据寄存器 | 未定义 |
| GPHPUD | 0x7F0080EC | 读/写 | 端口H上拉寄存器 | 0x00055555 |
| GPHCONSLP | 0x7F0080F0 | 读/写 | 端口H 睡眠模式配置寄存器 | 0x0 |
| GPHPUDSLP | 0x7F0080F4 | 读/写 | 端口H睡眠模式上拉/下拉寄存器 | 0x0 |

### 9. 端口I控制寄存器

端口I控制寄存器包括5个控制寄存器，如表4-14所示，分别是GPICON、GPIDAT、GPIPUD、GPICONSLP和GPIPUDSLP。

表4-14 端口I控制寄存器的5个控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPICON | 0x7F008100 | 读/写 | 端口I 配置寄存器 | 0x00 |
| GPIDAT | 0x7F008104 | 读/写 | 端口I数据寄存器 | 未定义 |
| GPIPUD | 0x7F008108 | 读/写 | 端口I上拉寄存器 | 0x55555555 |
| GPICONSLP | 0x7F00810C | 读/写 | 端口I睡眠模式配置寄存器 | 0x0 |
| GPIPUDSLP | 0x7F008110 | 读/写 | 端口I睡眠模式上拉/下拉寄存器 | 0x0 |

### 10. 端口J控制寄存器

端口J控制寄存器包括5个控制寄存器，如表4-15所示，分别是GPJCON、GPJDAT、GPJPUD、GPJCONSLP和GPJPUDSLP。

表4-15 端口J控制寄存器的5个控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPJCON | 0x7F008120 | 读/写 | 端口J 配置寄存器 | 0x00 |
| GPJDAT | 0x7F008124 | 读/写 | 端口J数据寄存器 | 未定义 |
| GPJPUD | 0x7F008128 | 读/写 | 端口J上拉寄存器 | 0x05555555 |
| GPJCONSLP | 0x7F00812C | 读/写 | 端口J睡眠模式配置寄存器 | 0x0 |
| GPJPUDSLP | 0x7F008130 | 读/写 | 端口J睡眠模式上拉/下拉寄存器 | 0x0 |

### 11. 端口K控制寄存器

端口K控制寄存器包括4个控制寄存器，如表4-16所示，分别是GPKCON0、GPKCON1、GPKDAT和GPKPUD。

表4-16 端口K控制寄存器的4个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPKCON0 | 0x7F008800 | 读/写 | 端口K 配置寄存器0 | 0x22222222 |
| GPKCON1 | 0x7F008804 | 读/写 | 端口K 配置寄存器1 | 0x22222222 |
| GPKDAT | 0x7F008808 | 读/写 | 端口K数据寄存器 | 未定义 |
| GPKPUD | 0x7F00880C | 读/写 | 端口K上拉/下拉寄存器 | 0x55555555 |

### 12. 端口L控制寄存器

端口L控制寄存器包括4个控制寄存器，如表4-17所示，分别是GPLCON0、GPLCON1、GPLDAT和GPLPUD。

表4-17 端口L控制寄存器的4个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPLCON0 | 0x7F008810 | 读/写 | 端口L 配置寄存器0 | 0x22222222 |
| GPLCON1 | 0x7F008814 | 读/写 | 端口L 配置寄存器1 | 0x22222222 |
| GPLDAT | 0x7F008818 | 读/写 | 端口L数据寄存器 | 未定义 |
| GPLPUD | 0x7F00881C | 读/写 | 端口L上拉/下拉寄存器 | 0x15555555 |

### 13. 端口M控制寄存器

端口M控制寄存器包括3个控制寄存器，如表4-18所示，分别是GPMCON、GPMDAT和GPMPUD。

表4-18 端口M控制寄存器的3个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPMCON | 0x7F008820 | 读/写 | 端口M 配置寄存器 | 0x22222222 |
| GPMDAT | 0x7F008824 | 读/写 | 端口M数据寄存器 | 未定义 |
| GPMPUD | 0x7F008828 | 读/写 | 端口M上拉/下拉寄存器 | 0x000002AA |

### 14. 端口N控制寄存器

端口N控制寄存器包括3个控制寄存器，如表4-19所示，分别是GPNCON、GPNDAT和GPNPUD。

表4-19 端口N控制寄存器的3个控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPNCON | 0x7F008830 | 读/写 | 端口N配置寄存器 | 0x00 |
| GPNDAT | 0x7F008834 | 读/写 | 端口N数据寄存器 | 未定义 |
| GPNPUD | 0x7F008838 | 读/写 | 端口N上拉/下拉寄存器 | 0x55555555 |

### 15. 端口O控制寄存器

端口O控制寄存器包括5个控制寄存器，如表4-20所示，分别是GPOCON、GPODAT、GPOPUD、GPOCONSLP和GPOPUDSLP。

**表4-20 端口O控制寄存器的5个控制寄存器**

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPOCON | 0x7F008140 | 读/写 | 端口O 配置寄存器 | 0xAAAAAAAA |
| GPODAT | 0x7F008144 | 读/写 | 端口O数据寄存器 | 未定义 |
| GPOPUD | 0x7F008148 | 读/写 | 端口O上拉寄存器 | 0x0 |
| GPOCONSLP | 0x7F00814C | 读/写 | 端口O睡眠模式配置寄存器 | 0x0 |
| GPOPUDSLP | 0x7F008150 | 读/写 | 端口O睡眠模式上拉/下拉寄存器 | 0x0 |

### 16. 端口P控制寄存器

端口P控制寄存器包括5个控制寄存器，如表4-21所示，分别是GPPCON、GPPDAT、GPPPUD、GPPCONSLP和GPPPUDSLP。

**表4-21 端口P控制寄存器的5个控制寄存器**

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPPCON | 0x7F008160 | 读/写 | 端口P配置寄存器 | 0x2AAAAAAA |
| GPPDAT | 0x7F008164 | 读/写 | 端口P数据寄存器 | 未定义 |
| GPPPUD | 0x7F008168 | 读/写 | 端口P上拉寄存器 | 0x1011AAA0 |
| GPPCONSLP | 0x7F00816C | 读/写 | 端口P睡眠模式配置寄存器 | 0x0 |
| GPPPUDSLP | 0x7F008170 | 读/写 | 端口P睡眠模式上拉/下拉寄存器 | 0x0 |

### 17. 端口Q控制寄存器

端口Q控制寄存器包括5个控制寄存器，如表4-22所示，分别是GPQCON、GPQDAT、GPQPUD、GPQCONSLP和GPQPUDSLP。

**表4-22 端口Q控制寄存器的5个控制寄存器**

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPQCON | 0x7F008180 | 读/写 | 端口Q 配置寄存器 | 0x0002AAAA |
| GPQDAT | 0x7F008184 | 读/写 | 端口Q数据寄存器 | 未定义 |
| GPQPUD | 0x7F008188 | 读/写 | 端口Q上拉寄存器 | 0x0 |
| GPQCONSLP | 0x7F00818C | 读/写 | 端口Q睡眠模式配置寄存器 | 0x0 |
| GPQPUDSLP | 0x7F008190 | 读/写 | 端口Q睡眠模式上拉/下拉寄存器 | 0x0 |

## 18. 特殊端口控制寄存器

特殊端口控制寄存器描述如表4-23、表4-24所示。

表4-23 特殊端口控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| SPCON | 0x7F0081A0 | 读/写 | 特殊端口控制寄存器 | 0xBFC11500 |

表4-24 特殊端口控制寄存器的位分配

| SPCON | 位 | 描述 | 初始状态 |
|---|---|---|---|
| DRVCON_CAM | [31:30] | CAMERA端口驱动<br>00=2mA 01=4 mA 10=7 mA 11=9 mA | 10 |
| DRVCON_HSSPI | [29:28] | HSSPI端口驱动<br>00=2mA 01=4 mA 10=7 mA 11=9 mA | 11 |
| DRVCON_HSMMC | [27:26] | HSMMCI端口驱动<br>00=2mA 01=4 mA 10=7 mA 11=9 mA | 11 |
| DRVCON_LCD | [25:24] | LCDI端口驱动<br>00=2mA 01=4 mA 10=7 mA 11=9 mA | 11 |
| DRVCON_MODEM | [23:22] | MODEM端口驱动<br>00=2mA 01=4 mA 10=7 mA 11=9 mA | 11 |
| Reserved | [21] | 保留 | 0 |
| nRSTOUT_OEM | [20] | 复位输出管脚（XnRSTOUT）输出使能<br>0=使能 1=不能 | 0 |
| DRVCON_SPICLK1 | [19:18] | SPICLK[1]端口驱动<br>00=2mA 01=4 mA 10=7 mA 11=9 mA | 00 |
| MEM1_DQS_PUD1 | [17:16] | 存储器端口1DWA管脚上拉/下拉控制<br>00=不能 01=下拉<br>10=上拉 11=保留 | 01 |
| MEM1_DQS_PUD0 | [13:12] | 存储器端口1数据管脚[31:16]上拉/下拉控制<br>00=不能 01=下拉<br>10=上拉 11=保留 | 01 |
| Reserved | [11:10] | 保留 | - |
| MEM1_DQS_PUD | [9:8] | 存储器端口0数据管脚[31:16]上拉/下拉控制<br>00=不能 01=下拉<br>10=上拉 11=保留 | 01 |
| USBH_DMPD | [7] | USB_Host DP下拉控制<br>0=disable 1=enable | 0 |
| USBH_DPPD | [6] | USB_Host DM 下拉控制<br>0=disable 1=enable | 0 |

（续）

| SPCON | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| USBH_PUSW2 | [5] | 当USBH_PUSW1 处于ON状态是USB_Host Pull-up 开关2是可控制的。<br>0=off　1=on | 0 |
| USBH_PUSW1 | [4] | USB_Host Pull-up 开关2控制<br>0=off　1=on | 0 |
| USBH_SUSPND | [3] | 使USB收发接口PAD进入暂停模式 | 0 |
| Reserved | [2] | 保留 | 0 |
| LCD_SEL | [1:0] | 选择LCD I/F管脚组态<br>00=Host I/F 形态　01=RGB I/F 形态<br>10=601/656 形态　11=保留 | 00 |

## 4.2.3 GPIO接口应用举例

下面介绍一个通过GPIO接口的GPK4～GPK7控制4个发光二极管LED1～LED4的编程实例，要求控制逻辑输出低电平时对应的发光二极管点亮。

### 1. 接口电路

4个发光二极管（LED）由4个GPIO口控制（GPK4～GPK7），接口电路如图4-3所示。

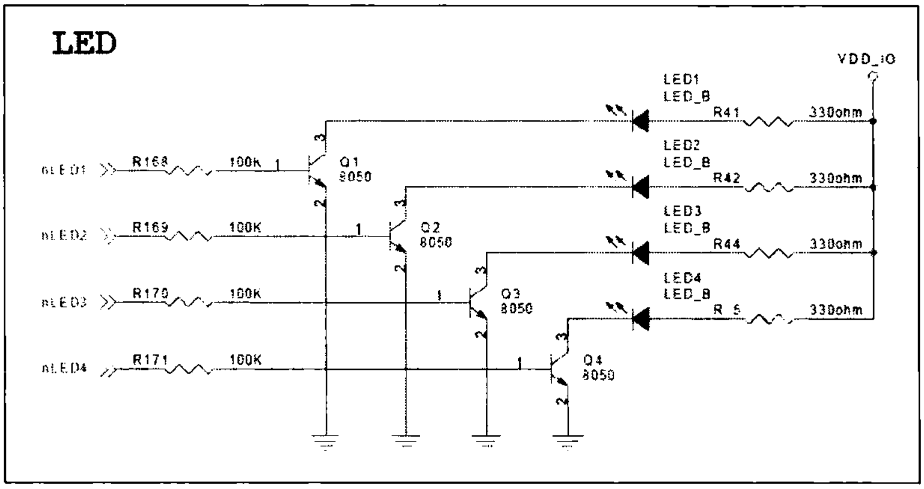



图4-3　4个发光二极管LED驱动电路

### 2. GPIO驱动程序

（1）宏定义

4个LED对应的宏定义如下：

```
#define LED_IOC_MAGIC      'c'
#define LED_IOC_MAXNR      6
#define LED_ON      _IO(LED_IOC_MAGIC, 1)
```

```
#define LED_OFF     _IO(LED_IOC_MAGIC, 2)
#define LED_1       _IO(LED_IOC_MAGIC, 3)
#define LED_2       _IO(LED_IOC_MAGIC, 4)
#define LED_3       _IO(LED_IOC_MAGIC, 5)
#define LED_4       _IO(LED_IOC_MAGIC, 6)
```

（2）初始化GPIO

设置GPIO为输出，将GPK4、GPK5、GPK6、GPK7设置为输出output=0001，即GPKCON0的19:28位都配置为0001；设置GPIO的数据，将GPKDATA的4:7位赋值为0。

```
 void led_init(void)
{
    iowrite32(((ioread32(S3C64XX_GPKCON) & 0x0000ffff) | 0x1111 << 16),S3C64XX_
GPKCON);
    iowrite16((ioread16(S3C64XX_GPKDAT) | 0xf0),S3C64XX_GPKDAT);
}
```

（3）实现LED驱动

判断是哪一个LED（arg参数指示LED灯的号码），判断类型是什么（点亮或熄灭），执行对应的指令，执行相应函数。

```
long GPIO_LED_ioctl(struct file *file,unsigned int cmd,unsigned long arg)
{
   if(_IOC_TYPE(cmd) != LED_IOC_MAGIC)
          return -EINVAL;
   if(_IOC_NR(cmd) > LED_IOC_MAXNR)
          return -EINVAL;
   switch(cmd) ) //辨识控制命令
   {
          case LED_OFF:
          iowrite16((ioread16(S3C64XX_GPKDAT) | 0x00f0),S3C64XX_GPKDAT);
                 break;
          case LED_ON:
       iowrite16((ioread16(S3C64XX_GPKDAT) & 0xff0f),S3C64XX_GPKDAT);
                 break;
          case LED_1:
                 gpio_set_value(S3C64XX_GPK(4),0);
                 break;
          case LED_2:
                 gpio_set_value(S3C64XX_GPK(5),0);
                 break;
          case LED_3:
                 gpio_set_value(S3C64XX_GPK(6),0);
                 break;
          case LED_4:
                 gpio_set_value(S3C64XX_GPK(7),0);
                 break;
          default:
                 printk("kernel:no cmd run \n");
                 return -EINVAL;
   }
   return 0;
}
```

# 4.3 PWM定时器

## 4.3.1 PWM定时器简介

S3C6410微处理器由5个32位定时器组成。这些定时器用来产生内部中断到ARM子系统。此外，定时器0、1、2和3包含一个PWM功能（脉宽调制），这个功能可以驱动外部的I/O信号。定时器0上的PWM能够产生一个可选的死区发生器，它被用来支持大量的通用装置。定时器4只是一个没有输出引脚的内部定时器，该定时器的时钟频率通常是APB-PCLK分频的版本。定时器0和1共享一个可编程的8位预定标器 ，它提供的第一层为PCLK。定时器2、3、4共享不同的8位预定标器，每个定时器拥有它自己的时钟分频器，个别时钟分频器提供有一个二级时钟版本（预定标器由2、4、8或16进行分频）。另外，定时器从外部引脚选择时钟来源。定时器0和1可以选择外部时钟TCLK0，定时器2、3和4可以选择外部时钟TCLK1。

每个定时器都有自己的32位减1计数器，由定时器时钟驱动。减1计数器用于初始化加载自定时器的计数缓冲寄存器（TCNTBn）。当减1计数器达到零时，定时器产生中断请求，通知CPU定时器操作完成。同时，相应的TCNTBn自动重新装入减1计数器的下一个周期的开始。如果定时器停止，例如，在定时器运行模式下，通过清除定时器的使能位TCONn，TCNTBn的值将不加载到计数器中。PWM使用TCMPBn寄存器的值，当减1计数器的值匹配于定时器控制器逻辑中的比较寄存器的值时，定时器控制器逻辑改变输出标准。因此，比较寄存器决定一个PWM输出的打开时间（或关闭时间）。TCNTBn和TCMPBn寄存器是双缓冲的，允许定时器参数在循环中更新，直到目前的定时器周期完成，新的值才能生效。

## 4.3.2 PWM的操作

### 1. 预定标器与分配器

一个8位预定标器和3位分配器产生以下输出频率，如表4-25所示。

表4-25 定时器最大、最小输出周期

| 4位分配器设置 | 最低分辨率（prescaler=0） | 最高分辨率（prescaler=255） | 最大间隔（TCNTBn=65535） |
|---|---|---|---|
| 1/1（PCLK=66MHz） | 0.015μs（66.0MHz） | 3.87μs（258kHz） | 0.23s |
| 1/2（PCLK=66MHz） | 0.031μs（33.0MHz） | 7.75μs（129kHz） | 0.50s |
| 1/4（PCLK=66MHz） | 0.060μs（16.5MHz） | 15.5μs（64.5kHz） | 1.02s |
| 1/8（PCLK=66MHz） | 0.121μs（8.25MHz） | 31.0μs（32.2kHz） | 2.03s |
| 1/16（PCLK=66MHz） | 0.242μs（4.13MHz） | 62.1μs（16.1kHz） | 4.07s |

### 2. 定时器的基本操作

定时器的基本操作流程如图4-4所示。

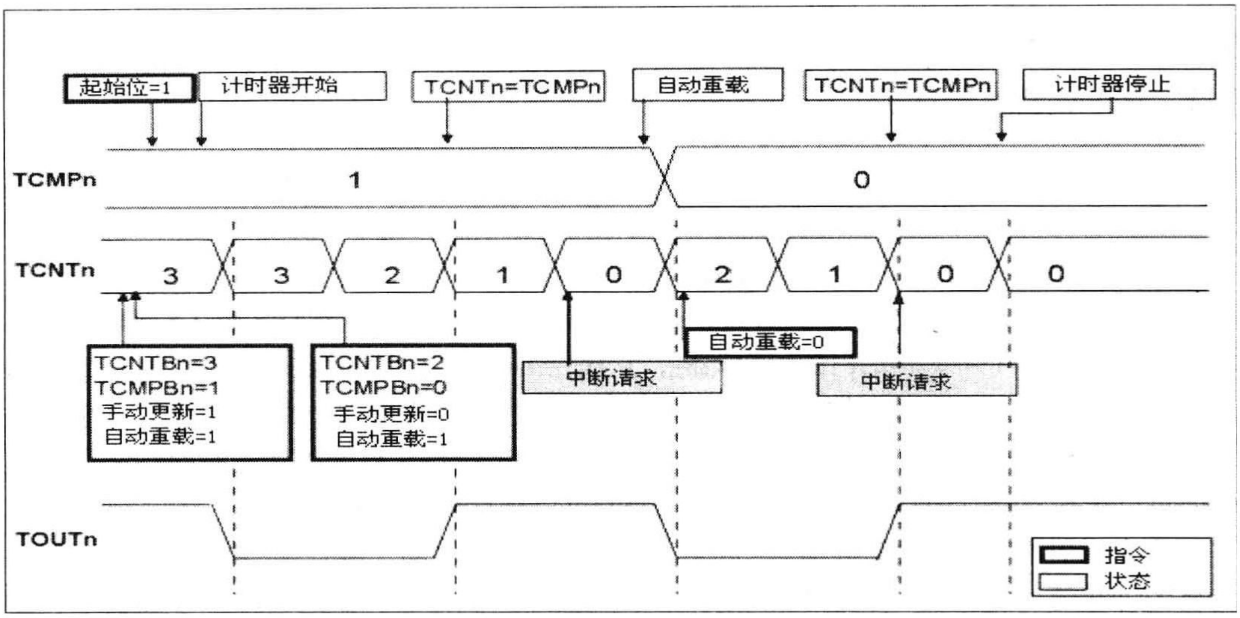



图4-4 定时器的操作

定时器（除定时器通道5）包括TCNTBn、TCNTn、TCMPBn和TCMPn。当定时器置0时，TCNTBn和TCMPBn载入TCNTn和TCMPn中。当TCNTn置0时，如果中断信号启动，则将产生中断请求。TCNTn和TCMPn是内部寄存器的名称，可以从TCNTBn寄存器中读取TCNTn寄存器。

### 3. 双缓冲和自动重新加载

定时器有一个双缓冲功能，在没有停止当前定时器操作基础上，它可以改变加载数值以适合下一定时器的操作。虽然新的定时器值被设定，但当前定时器的操作已经成功完成。定时器的值可以被写入TCNTBn（定时器计数缓冲寄存器），定时器当前计数器的值已经从TCNTBn中被读取（定时器计数观察寄存器）。如果读TCNTBn，这个值是下一个定时器的重载值不是当前计数器的状态。自动重新载入是一个操作，当TCNTn置0时，它复制TCNTBn到TCNTn。值写入TCNTBn，当TCNTn达到0并自动重新启动时，它只能被加载到TCNTn中。

### 4. 初始化定时器（设置手动向上数据和逆变器）

因为计数器达到0时，定时器发生自动重载，所以用户必需首先定义TCNTn的开始值。在这种情况下，自动更新位必须载入初始值。可以采取下列步骤启动定时器：

1）将初始值写到TCNTBn和TCMPBn中。

2）设置相应定时器的手动更新位（建议设置逆变器的开/关位，即是否用逆变器）。

3）设置相应定时器的起始位去启动定时器，并清空手动更新位。

### 5. PWM（脉宽调制）

PWM功能应执行TCMPBn。TCNTBn决定PWM的频率，TCMPBn决定PWM的值。若有较

高的PWM值，则减少TCMPBn值；若有较低的PWM值，则增加TCMPBn值。如果逆变器输出启用，递增/递减可能会适得其反，由于双缓冲的特性，对于下一个PWM周期，TCMPBn可以通过ISR被写入当前PWM周期的任何一端。

### 6. 输出电平控制

下列方法用来控制TOUT，作为高电平或低电平。（假定逆变器是关闭状态）

- 关闭自动重新载入位。然后TOUTn达到高电平，同时定时器被停止，之后TCNTn达到0。建议用这种方法。
- 通过清空定时器的开始/停止位为0来停止定时器。如果TCNTn<=TCMPn，则输出高电平。如果TCNTn>TCMPn，则输出低电平。
- 在TCON中，通过逆变器的开/关位TOUTn可以被转换。逆变器移除额外的电路以调节输出电平。

### 7. 死区发生器

该死区是用于电源设备的PWM控制。这种功能用于在关闭一个开关设备和打开另一个开关设备之间插入。这个定时器禁止两个开关设备同时转向，即很短的时间。其死区特性开启时输出波形比较。

### 8. DMA请求模式

在减1计数器的周期结束时，代替发送中断，定时器可以配置成发送一个DMA请求信号到DMA的一个通道中，这种模式是让DMA传输在源和目标之间在固定的时间段内发生。

DMA请求过程：定时器可以在任意时间产生DMA请求，并且保持DMA请求信号（nDMA_REQ）为低，直到定时器收到ACK信号。当定时器收到ACK信号时，它使请求信号变得无效。只有一个定时器的时间可以被设定为DMA请求源，设置DMA模式位（在TCFG1寄存器中）决定了定时器产生DMA请求。如果DMA请求模式设定一个特定的定时器，即定时器发送DMA请求，并产生正常的ARM中断请求。而其他的定时器未被设定为DMA模式，仍然可以生成正常的ARM中断。如果没有的计时器被设定在DMA模式，那么它们都可以产生正常的中断。

### 9. 定时器中断的产生

通过控制INTRGEN_SEL端口状态，PWMTIMER提供产生脉冲中断和电平中断的灵活性。当INTRGEN_SEL端口状态是逻辑1时，将产生可选电平或可选脉冲中断。在PWMTIMER内部写入具体值到TINT_CSTAT寄存器，控制中断产生。中断的产生是基于TINT_CSTAT寄存器中设置的值。

## 4.3.3 PWM专用寄存器

表4-26列出了定时器的专用寄存器。

表4-26 专用寄存器

| 寄存器 | 偏 移 量 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| TCFG0 | 0x7F006000 | 读/写 | 定时器配置寄存器0时配置两个8位<br>预定标器和死区长度 | 0x0000_0101 |
| TCFG1 | 0x7F006004 | 读/写 | 定时 器配置 寄存器1时，4-MUX和DMA模式选择寄存器 | 0x0000_0000 |
| TCON | 0x7F006008 | 读/写 | 定时器控制寄存器 | 0x0000_0000 |
| TCNTB0 | 0x7F00600C | 读/写 | 定时器0计数缓冲器 | 0x0000_0000 |
| TCMPB0 | 0x7F006010 | 读/写 | 定时器0比较缓冲寄存器 | 0x0000_0000 |
| TCNTO0 | 0x7F006014 | 读 | 定时器0计数观察寄存器 | 0x0000_0000 |
| TCNTB1 | 0x7F006018 | 读/写 | 定时器1计数缓冲器 | 0x0000_0000 |
| TCMPB1 | 0x7F00601c | 读/写 | 定时器1比较缓冲寄存器 | 0x0000_0000 |
| TCNTO1 | 0x7F006020 | 读 | 定时器1计数观察寄存器 | 0x0000_0000 |
| TCNTB2 | 0x7F006024 | 读/写 | 定时器2计数缓冲器 | 0x0000_0000 |
| TCMPB2 | 0x7F006028 | 读/写 | 定时器2比较缓冲寄存器 | 0x0000_0000 |
| TCNTO2 | 0x7F00602c | 读 | 定时器2计数观察寄存器 | 0x0000_0000 |
| TCNTB3 | 0x7F006030 | 读/写 | 定时器3计数缓冲器 | 0x0000_0000 |
| TCMPB3 | 0x7F006034 | 读/写 | 定时器3比较缓冲寄存器 | 0x0000_0000 |
| TCNTO3 | 0x7F006038 | 读 | 定时器3计数观察寄存器 | 0x0000_0000 |
| TCNTB4 | 0x7F00603c | 读/写 | 定时器4计数缓冲器 | 0x0000_0000 |
| TCNTO4 | 0x7F006040 | 读 | 定时器4计数观察寄存器 | 0x0000_0000 |
| TINT_CSTAT | 0x7F006044 | 读/写 | 定时器中断控制和状态寄存器 | 0x0000_0000 |

### 1. TCFG0（定时器配置寄存器0）

TCFG0（定时器配置寄存器0）的描述如表4-27、表4-28所示。

表4-27 TCFG0（定时器配置寄存器0）

| 寄存器 | 偏 移 量 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| TCFG0 | 0x7F006000 | 读/写 | 定时器配置寄存器0，可配置两个8位预定标器和死区长度 | 0x0000_0101 |

表4-28 TCFG0（定时器配置寄存器0）的位分配

| TCFG0 | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Reserved | [31:24] | 读 | 保留 | 0x00 |
| Dead zone length | [23:16] | 读/写 | 死区的长度 | 0x00 |
| Prescaler 1 | [15:8] | 读/写 | 预定标器1的值，用于定时器2、3和4 | 0x01 |
| Prescaler 0 | [7:0] | 读/写 | 预定标器0的值，用于定时器0和1 | 0x01 |

### 2. TCFG1（定时器配置寄存器1）

TCFG1（定时器配置寄存器1）的描述如表4-29、表4-30所示。

表4-29　TCFG1（定时器配置寄存器1）

| 寄存器 | 地　址 | 读/写 | 描　述 | 复 位 值 |
|---|---|---|---|---|
| TCFG1 | 0x7F006004 | 读/写 | 定时器配置寄存器1，可控制5MUX和DMA0 | x0000_0000 |

表4-30　TCFG1（定时器配置寄存器1）的位分配

| TCFG1 | 位 | 读/写 | 描　述 | 初始状态 |
|---|---|---|---|---|
| Reserved | [31:24] | 读 | 保留位 | 0x00 |
| DMA mode | [23:20] | 读/写 | 选择DMA请求通道选择位：<br>0000：无选择　0001：INT0<br>0010：INT1　0011：INT2<br>0100：INT 3　0101：INT4 | 0x00 |
| Divider MUX 4 | [19:16] | 读/写 | 选择定时器4的MUX输入：<br>0000：1/1　0001：1/2<br>0010：1/4　0011：1/8<br>0100：1/16　0101：外部TCLK1 | 0x00 |
| Divider MUX 3 | [15:12] | 读/写 | 选择定时器3的MUX输入：<br>0000：1/1　0001：1/2<br>0010：1/4　0011：1/8<br>0100：1/16　0101：外部TCLK1 | 0x00 |
| Divider MUX 2 | [11:8] | 读/写 | 选择定时器2的MUX输入：<br>0000：1/1　0001：1/2<br>0010：1/4　0011：1/8<br>0100：1/16　0101：外部TCLK1 | 0x00 |
| Divider MUX 1 | [7:4] | 读/写 | 选择定时器1的MUX输入：<br>0000：1/1　0001：1/2<br>0010：1/4　0011：1/8<br>0100：1/16　0101：外部TCLK1 | 0x00 |
| Divider MUX 0 | [3:0] | 读/写 | 选择定时器0的MUX输入：<br>0000：1/1　0001：1/2<br>0010：1/4　0011：1/8<br>0100：1/16　0101：外部TCLK1 | 0x00 |

### 3. TCON（定时控制寄存器）

TCON（定时控制寄存器）的描述如表4-31和表4-32所示。

表4-31 TCON（定时控制寄存器）

| 寄存器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| TCON | 0x7F006008 | 读/写 | 定时器控制寄存器 | 0x0000_0000 |

表4-32 TCON（定时控制寄存器）的位分配

| TCON | 位 | 读/写 | 描 述 | 初始值 |
|---|---|---|---|---|
| Reserved | [31:23] | 读 | 保留 | 0x000d |
| Timer 4 Manual Update（note） | [21] | 读/写 | 确定定时器4的手动更新<br>0 = 无操作 1 = 更新TCNTB4 | 0x0 |
| Timer 4 Start/Stop | [20] | 读/写 | 确定定时器4的启动/停止<br>0 = 停止 1 = 开始定时器4 | 0x0 |
| Timer 3 Auto Reload on/off | [19] | 读/写 | 确定定时器3的自动加载开/关<br>0 = One-shot 1 =间隔模式（自动重载） | 0x0 |
| Timer 3 output inverter on/off | [18] | 读/写 | 确定定时器3的输出反转器开/关<br>0 = 逆变器关闭 1 = 逆变器开，用于TOUT3 | 0x0 |
| Timer 3 Manual Update（note） | [17] | 读/写 | 确定定时器3的手动更新<br>0 = 无操作 1=更新TCNTB3或TCMPB3 | 0x0 |
| Timer 3 Start/Stop | [16] | 读/写 | 确定定时器3的启动/停止<br>0 = 停止 1 = 开始定时器3 | 0x0 |
| Timer 2 Auto Reload on/off | [15] | 读/写 | 确定定时器2的自动加载开/关<br>0 = One-shot 1 =间隔模式（自动重载） | 0x0 |
| Timer 2 Output Inverter on/off | [14] | 读/写 | 确定定时器2的输出反转器开/关<br>0 =逆变器关闭1 =逆变器开，用于TOUT2 | 0x0 |
| Timer 2 Manual Update（note） | [13] | 读/写 | 确定定时器2的手动更新<br>0 =无操作1 = 更新TCNTB2或TCMPB2 | 0x0 |
| Timer 2 Start/Stop | [12] | 读/写 | 确定定时器2的启动/停止<br>0 = 停止 1 =开始定时器2 | 0x0 |
| Timer 1 Auto Reload on/off | [11] | 读/写 | 确定定时器1的自动加载开/关<br>0 = One-shot 1 =间隔模式（自动重载） | 0x0 |
| Timer 1 Output Inverter on/off | [10] | 读/写 | 确定定时器1的输出反转器开/关<br>0 =逆变器关闭1 = 逆变器开，用于TOUT1 | 0x0 |
| Timer 1 Manual Update（note） | [9] | 读/写 | 确定定时器1的手动更新<br>0 =无操作1 = 更新TCNTB1或TCMPB1 | 0x0 |
| Timer 1 Start/Stop | [8] | 读/写 | 确定定时器1的启动/停止<br>0=停止1=开始定时1 | 0x0 |
| Reserved | [7:5] | 读/写 | 保留 | 0x0 |

（续）

| TCON | 位 | 读/写 | 描　述 | 初始值 |
|---|---|---|---|---|
| Dead Zone Enable | [4] | 读/写 | 确定死区的操作<br>0=禁止 1=使能 | 0x0 |
| Timer 0 Auto Reload on/off | [3] | 读/写 | 确定定时器0的自动加载开/关<br>0=One-shot 1=间隔模式（自动重载） | 0x0 |
| Timer 0 output inverter on/off | [2] | 读/写 | 确定定时器0的输出反转器开/关<br>0=逆变器关闭 1=逆变器开，用于TOUT0 | 0x0 |
| Timer 0 Manual Update （note） | [1] | 读/写 | 确定定时器0的手动更新<br>0 =无操作1 = 更新TCNTB0或TCMPB0 | 0x0 |
| Timer 0 Start/Stop | [0] | 读/写 | 确定定时器0的启动/停止<br>0 = 停止 1 = 开始定时器0 | 0x0 |

### 4. TCNTB0（定时器0计数寄存器）

TCNTB0（定时器0计数寄存器）的描述如表4-33、表4-34所示。

表4-33　TCNTB0（定时器0计数寄存器）

| 寄存器 | 地　址 | 读/写 | 描　述 | 复 位 值 |
|---|---|---|---|---|
| TCNTB0 | 0x7F00600C | 读/写 | 定时器0计数缓冲器 | 0x00000000 |

表4-34　TCNTB0（定时器0计数寄存器）的位分配

| TCNTB0 | 位 | 读/写 | 描　述 | 初始状态 |
|---|---|---|---|---|
| Timer 0 Count Buffer | [31:0] | 读/写 | 设置定时器0的计数缓冲器的值 | 0x00000000 |

### 5. TCMPB0（定时器0比较寄存器）

TCMPB0（定时器0比较寄存器）的描述如表4-35、表4-36所示。

表4-35　TCMPB0（定时器0比较寄存器）

| 寄存器 | 地　址 | 读/写 | 描　述 | 复 位 值 |
|---|---|---|---|---|
| TCMPB0 | 0x7F006010 | 读/写 | 定时器0比较缓冲寄存器 | 0x0000_0000 |

表4-36　TCMPB0定时器0比较寄存器的位分配

| TCMPB0 | 位 | 读/写 | 描　述 | 初始状态 |
|---|---|---|---|---|
| Timer 0 Compare Buffer | [31:0] | 读/写 | 设置定时器0的比较缓冲器的值 | 0x00000000 |

### 6. TCNTO0（定时器0计数观察寄存器）

TCNTO0（定时器0计数观察寄存器）的描述如表4-37、表4-38所示。

表4-37 TCNTO0定时器0计数观察寄存器

| 寄存器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| TCNTO0 | 0x7F006014 | 读 | 定时器0计数观察寄存器 | 0x0000_0000 |

表4-38 TCNTO0定时器0计数观察寄存器

| TCNTO0 | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Timer 0 Count Observation | [31:0] | 读 | 设置定时器0计数观察寄存器的值 | 0x00000000 |

### 7. TINT_CSTAT（定时器中断控制和状态寄存器）

TINT_CSTAT（定时器中断控制和状态寄存器）的描述如表4-39、表4-40所示。

表4-39 TINT_CSTAT（定时器中断控制和状态寄存器）

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| INT_CSTAT | 0x7F006044 | 读/写 | 定时器中断控制和状态寄存器 | x0000_0000 |

表4-40 TINT_CSTAT（定时器中断控制和状态寄存器）的位分配

| TINT_CSTAT | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Reserved | [31:10] | 读 | 保留位 | 0x00000 |
| Timer 4 Interrupt Status | [9] | 读/写 | 定时器4中断状态位。通过写1清除该位 | 0x0 |
| Timer 3 Interrupt Status | [8] | 读/写 | 定时器3中断状态位。通过写1清除该位 | 0x0 |
| Timer 2 Interrupt Status | [7] | 读/写 | 定时器2中断状态位。通过写1清除该位 | 0x0 |
| Timer 1 Interrupt Status | [6] | 读/写 | 定时器1中断状态位。通过写1清除该位 | 0x0 |
| Timer 0 Interrupt Status | [5] | 读/写 | 定时器0中断状态位。通过写1清除该位 | 0x0 |
| Timer 4 interrupt Enable | [4] | 读/写 | 定时器4中断启动。1：启动 0：禁止 | 0x0 |
| Timer 3 interrupt Enable | [3] | 读/写 | 定时器3中断启动。1：启动 0：禁止 | 0x0 |
| Timer 2 interrupt Enable | [2] | 读/写 | 定时器2中断启动。1：启动 0：禁止 | 0x0 |
| Timer 1 interrupt Enable | [1] | 读/写 | 定时器1中断启动。1：启动 0：禁止 | 0x0 |
| Timer 0 interrupt Enable | [0] | 读/写 | 定时器0中断启动。1：启动 0：禁止 | 0x0 |

## 4.3.4 定时器应用举例

使用S3C6410通过PWM控制蜂鸣器，连接蜂鸣器的PWM0对应GPF14，该引脚可通过软件设置为PWM输出，也可以作为普通的GPIO使用。

### 1. 接口电路

通过PWM控制蜂鸣器的接口电路如图4-5所示。

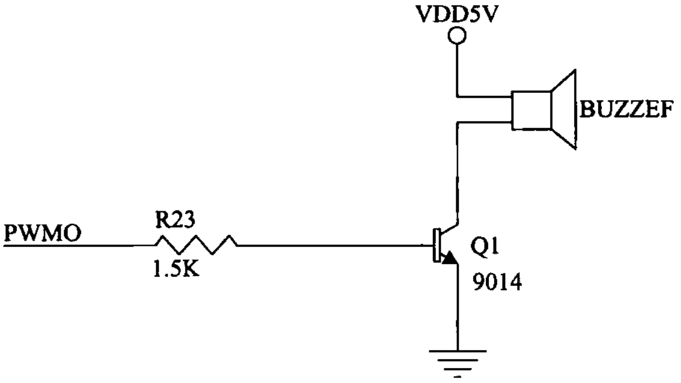



图4-5 蜂鸣器接口电路

### 2. 蜂鸣器驱动程序

首先把GPF14端口设置为PWM功能输出，再设定相应的定时器就可以控制PWM的输出频率了。驱动代码如下：

```
#define DEVICE_NAME     "pwm"      //定义设备名
#define PWM_IOCTL_SET_FREQ         1  //定义宏变量,用于后面的ioctl中的switch case
#define PWM_IOCTL_STOP             0  //定义信号量lock
static struct semaphore lock;

/* freq:  pclk/50/16/65536 ~ pclk/50/16
  * if pclk = 50MHz,freq is 1Hz to 62500Hz
  * human ear : 20~ 20000Hz
  */
static void PWM_Set_Freq(unsigned long freq)  //设置PWM的频率,配置各寄存器
{
   unsigned long tcon;
   unsigned long tcnt;
   unsigned long tcfg1;
   unsigned long tcfg0;
   struct clk *clk_p;
   unsigned long pclk;
   unsigned tmp;

   tmp = readl(S3C64XX_GPFCON);           //设置GPF14为TOUT0,PWM输出
   tmp &= ~(0x3U << 28);
   tmp |=  (0x2U << 28);
   writel(tmp,S3C64XX_GPFCON);
   tcon = __raw_readl(S3C_TCON);          //读定时器配置寄存器TCON到tcon
   tcfg1 = __raw_readl(S3C_TCFG1);        //读寄存器TCFG1到tcfg1
   tcfg0 = __raw_readl(S3C_TCFG0);        //读寄存器TCFG0到tcfg0

   //prescaler = 50
   tcfg0 &= ~S3C_TCFG_PRESCALER0_MASK;
   //S3C_TCFG_PRESCALER0_MASK=255(11111111),为定时器0和1的预分频值的掩码,TCFG[0~8]
   tcfg0 |= (50 - 1);                     //tcfg0=00110001,预分频为50
   //mux = 1/16
   tcfg1 &= ~S3C_TCFG1_MUX0_MASK;         //定时器0分割值的掩码 (15<<0)
   tcfg1 |= S3C_TCFG1_MUX0_DIV16;         //定时器0进行1/16分割(4<<0)->0100
```

```
    __raw_writel(tcfg1,S3C_TCFG1);          //将tcfg1的值写到分割寄存器中
    __raw_writel(tcfg0,S3C_TCFG0);          //将tcfg0的值写到分频寄存器中
    clk_p = clk_get(NULL, "pclk");          //得到pclk
    pclk  = clk_get_rate(clk_p);
    tcnt  = (pclk/50/16)/freq;              //得到定时器的输入时钟,设置PWM的调制频率
    __raw_writel(tcnt,S3C_TCNTB(0));        //PWM的频率等于定时器的输入时钟
    __raw_writel(tcnt/2,S3C_TCMPB(0));      //占空比是50%

    tcon &= ~0x1f;
    tcon |= 0xb;            //禁用死区, 间隔模式开启, 逆变器关闭, 自动更新TCNTB0&TCMPB0,
                            //开始定时器0
    __raw_writel(tcon,S3C_TCON);      //把tcon的设置写到计数控制寄存器S3C_TCON中
                                      //tcon &= ~2;
    __raw_writel(tcon,S3C_TCON);
}

void PWM_Stop(void)
{
    unsigned tmp;
    tmp = readl(S3C64XX_GPFCON);            //设置GPF14为输出
    tmp &= ~(0x3U << 28);
    writel(tmp,S3C64XX_GPFCON);             //设置GPF14为低电平,使蜂鸣器停止
}

static int s3c64xx_pwm_open(struct inode *inode,struct file *file)
{
    if(!down_trylock(&lock))                //如果是,down_trylock(&lock)=0,否则非0
            return 0;
    else
            return -EBUSY;                  //返回错误信息: 请求资源不可用
}

static int s3c64xx_pwm_close(struct inode *inode,struct file *file)
{
    up(&lock);
    return 0;
}

/*cmd是1,表示设置频率;cmd是0,表示停止pwm*/
static long s3c64xx_pwm_ioctl(struct file *filep,unsigned int cmd,unsigned long arg)
{
    switch(cmd) {  //如果cmd=1,进入case  PWM_IOCTL_SET_FREQ
            case PWM_IOCTL_SET_FREQ:
                    if(arg == 0)            //如果设置的频率参数是0
                    return -EINVAL;         //返回错误信息,表示向参数传递了无效的参数
                    PWM_Set_Freq(arg);      //否则设置给定的频率
                    break;
            case PWM_IOCTL_STOP:            //如果cmd=0,进入
            default:
                    PWM_Stop();             //停止蜂鸣器
                    break;
    }
    return 0;
}
```

# 4.4　DMA控制器

## 4.4.1　DMA控制器简介

S3C6410包含4个的DMA控制器，每个DMA控制器由8个传输的通道组成。DMA控制器的每个通道能在SPINEAXI总线的设备和/或者PERIPHERALAXI总线之间通过AHB到AXI（没有其他限制）进行数据传输。每个通道可以处理以下4种情况：

- 源及目标在中心总线上；
- 当目标在外设总线上可用时，在中心总线上，源也可用；
- 当目标在中心总线上可用时，在外设总线上，源也可用；
- 源及目标在外设总线上可用。

### 1. DMA控制器

DMA控制器的结构框图如图4-6所示。

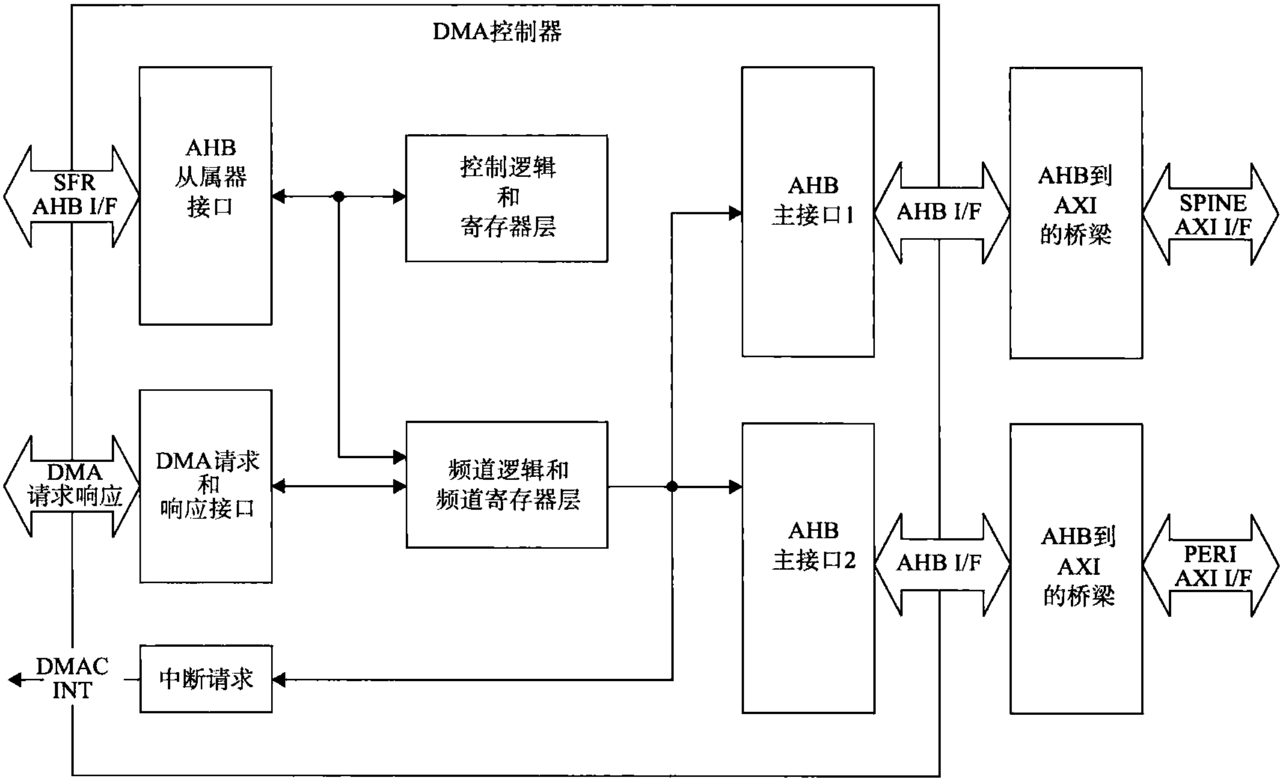



图4-6　DMA控制器的结构框图

### 2. DMA接口

DMA控制器支持4种类型的传输：

- 从内存到外设；
- 从外设到内存；
- 从内存到内存；

● 从外设到外设。

每一种传输类型可以转让任一外设，或DMA控制器作为流量控制器。因此，有4种可能的控制情况。

（1）在DMA控制器的流控制下外设到内存的处理

对于不是脉冲大小倍数的处理，使用脉冲和单一请求信号，如图4-7所示。

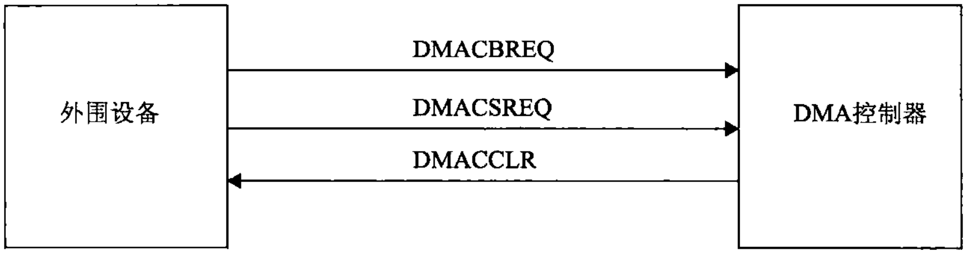



图4-7 由脉冲和单一请求组成的外设到内存的处理

这两个请求信号并非互相排斥的，DMA控制器监控DMACBREQ，数据的传输数量大于脉冲大小时，并当发生请求时，开始一个脉冲传输（来自外设）。当数据的传输数量小于脉冲大小时，DMA控制器监控DMACBREQ，并当发生请求时，开始单一传输。

（2）在DMA控制器的流控制下内存到外设的处理

处理多种模块大小，只用于脉冲模块请求信号，如图4-8所示，由脉冲组成的内存到外设的处理。

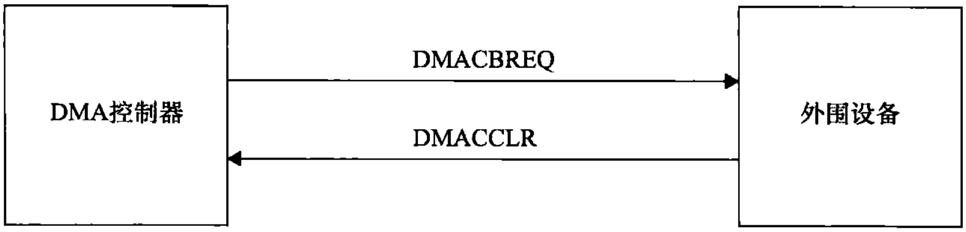



图4-8 由脉冲组成的内存到外设的处理

只请求DMACBREQ。当剩余的数据数量大于脉冲的大小时，DMA控制器发送数据的全脉冲。当剩余的数据数量小于脉冲的大小时，DMAC再次监控DMACBREQ，并且当发生请求的时候，传输剩余的数据。

（3）在DMA控制器的流控制下内存到内存的处理

当启用软件程序从内存到内存传输的DMA通道时，DMA通道中没有DMA请求开始传输，直到发生下列情况中的一种：

● 所有数据转移；
● 通过软件禁止该通道。

必须执行内存到内存的传输与低通道优先，否则，其他DMA通道不能进入总线，直到内

存到内存的传输已经完成，或其他AHB的控制无法执行任何处理。

内存到内存的处理示意如图4-9所示。

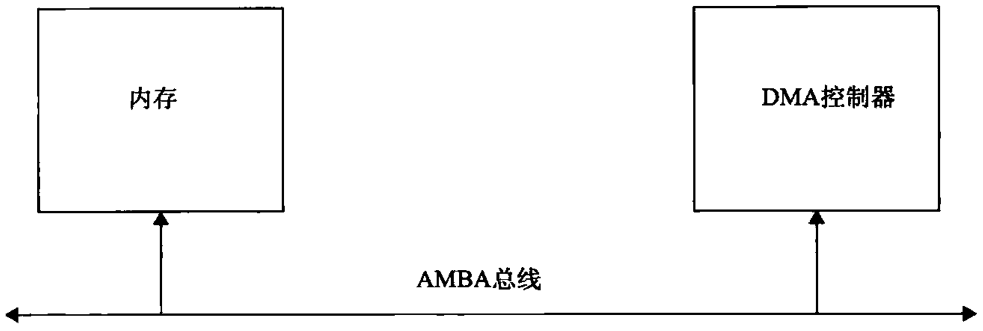



图4-9 在DMA控制器的流控制下内存到内存的处理

（4）在DMA控制器的流控制下外设到外设的处理

当处理的不是脉冲大小的倍数时，用下面的信号：

- 单一和脉冲请求（DMACBREQandDMACSREQ），来源是外设信号。
- 脉冲请求（DMACBREQ），目标是外设信号。

外设到外设的处理示意如图4-10所示。

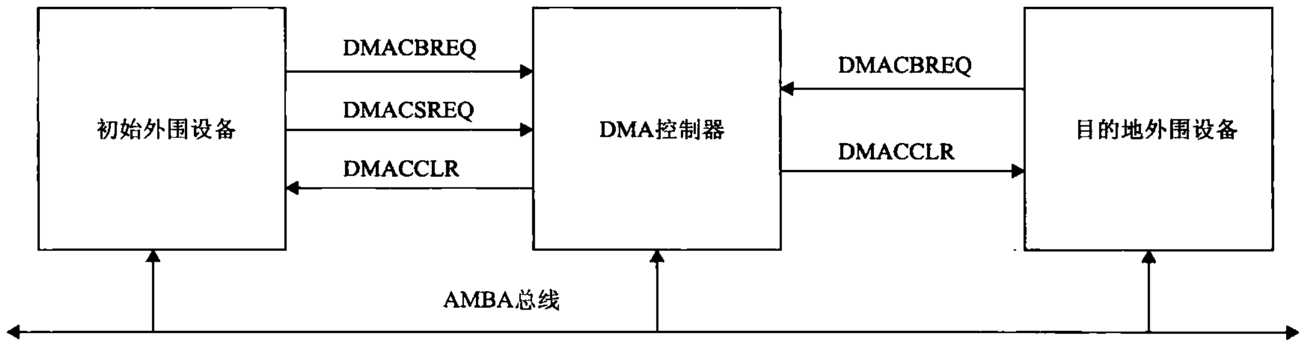



图4-10 由脉冲和单一请求组成的外设到外设的处理

源外设遵循同样的程序，可当做描述外设到内存的DMA控制器的流量控制。目标外设遵循同样的程序，可当做描述内存到外设的处理下外设的流量控制。下一个LLI装载时，所有读与写的传输是完整的。可以使用DMACTC的信号表明，过去的数据已传输到外设上，如表4-41所示。

表4-41 DMA控制器的流量控制

| 传 输 方 向 | 请求发生器 | 请求信号使用 |
| --- | --- | --- |
| 外设到内存 | 外设 | DMACBREQ |
| 内存到外设 | 外设 | DMACBREQ |

（续）

| 传 输 方 向 | 请求发生器 | 请求信号使用 |
|---|---|---|
| 内存到内存 | DMA控制器 | None |
| 外设到外设 | 外设 | Src：DMACBREQ |

## 4.4.2 DMA专用寄存器

有4个DMA控制器，分别为DMAC0、DMAC1、SDMAC0和SDMAC1。DMAC0、DMAC1、SDMAC0和SDMAC1寄存器基地址分别是0x7500_0000、0x7510_0000、0x7DB0_0000和0x7DC0_0000。

用于OneNAND控制器的页面访问功能是通过DMAC0和SDAMC0的通道3添加的。DMA寄存器的位置以及对DMA寄存器进行的描述如下所示。

### 1. 配置寄存器DMACConfiguration

DMACConfiguration读/写寄存器，用于配置DMA控制器的操作。个别AHB主接口的字节序，可以通过写入这个寄存器的M1和M2被改变。M1允许AHB主机口1的字节序被改变，M2允许AHB主机口2的字节序被改变。AHB主机口在复位上被设置为小端模式。表4-42所示为DMACConfiguration寄存器的位分配。

表4-42 DMACConfiguration寄存器的位分配

| DMACConfiguration | 位 | 读/写 | 功 能 |
|---|---|---|---|
| M2 | [2] | 读/写 | AHB主接口2字节序配置：<br>0 =小端模式<br>1 =大端模式<br>这个位复位为0 |
| M1 | [1] | 读/写 | AHB主接口1字节序配置：<br>0 =小端模式<br>1 =大端模式<br>这个位复位为0 |
| E | [0] | 读/写 | DMA控制器启动：<br>0 =禁止<br>1 =启动<br>这个位复位为0，禁止DMA控制器还原电力消耗 |

### 2. 同步寄存器DMACSync

DMACSync读/写寄存器，用于为DMA请求信号启动/禁止同步逻辑。DMA请求信号由DMACBREQ[15:0]、DMACSREQ[15:0]、DMACLBREQ[15:0]和DMACLSREQ[15:0]信号组成。位设置为0，用于DMA请求的特殊组启动同步逻辑。位设置为1，用于DMA请求的特殊

组禁止同步逻辑。这个寄存器复位为0，同步逻辑启动。表4-43所示为DMACSync寄存器的位分配。

表4-43 DMACSync寄存器的位分配

| DMACSync | 位 | 读/写 | 功　能 |
|---|---|---|---|
| DMACSync | [15:0] | 读/写 | 用于DMA请求信号启动或禁止的DMA同步逻辑。低位指示，同步逻辑用于DMACBREQ[15:0]，DMACSREQ[15:0]，DMACLBREQ[15:0]，和DMACLSREQ[15:0]请求信号的启 动。高位指 示，同步逻辑禁止 |

### 3. 通道源地址寄存器DMACCxSrcAddr

8位读/写DMACCxSrcAddr寄存器包含当前数据的源地址（字节对齐）。通道启动前，每个寄存器都直接通过软件编程。当DMA通道启动这个寄存器，通道源地址寄存器是加1递增。

当通道有效，但不提供有用的信息时，读这个寄存器。这是因为通过时间，软件处理读取值，通道可能有进步。当通道停止时，只能故意被读取，在这种情况下，它显示最后信息，源地址被读取。表4-44所示为DMACCxSrcAddr寄存器的位分配。

表4-44 DMACCxSrcAddr寄存器的位分配

| DMACCxSrcAddr | 位 | 读/写 | 功　能 |
|---|---|---|---|
| SrcAddr | [31:0] | 读/写 | DMA源地址 |

### 4. 通道目标地址寄存器DMACCxDestAddr

8位读/写DMACCxDestAddr寄存器包含当前数据的源地址（字节定位）。通道启动前，每个寄存器都直接通过软件编程。当DMA通道启动时，随着目标地址的增加，该寄存器不断更新。

当通道有效，但不提供有用的信息时，读这个寄存器。这是因为通过时间，软件处理读取值，通道可能有进步。当通道停止时，只能故意被读取，在这种情况下，它显示最后信息，源地址被读取。表4-45所示为DMACCxDestAddr寄存器的位分配。

表4-45 DMACCxDestAddr寄存器的位分配

| DMACCxDestAddr | 位 | 读/写 | 功　能 |
|---|---|---|---|
| DestAddr | [31:0] | 读/写 | DMA目标地址 |

### 5. 通道链表列表项目寄存器DMACCxLLI

8位读/写DMACCxLLI寄存器包含下一个链表列表项目（LLI）的字对齐地址。如果还有LLI，当前LLI为最后一个时，并一旦所有的DMA发送完成，DMA通道被禁止。表4-46所示为DMACCxLLI寄存器的位分配。

表4-46 DMACCxLLI寄存器的位分配

| DMACCxLLI | 位 | 读/写 | 功 能 |
|---|---|---|---|
| LLI | [31:2] | 读/写 | 链表列表项目<br>用于下一个LLI，地址的位[31:2]<br>地址的位[1:0] |
| R | [1] | 读/写 | 保留，必须被写入为0，屏蔽读 |
| LM | [0] | 读/写 | 用于下载下一个LLI的AHB主选择：<br>LM = 0 = AHB主机1<br>LM = 1 = AHB主机2 |

### 6. 通道控制寄存器DMACCxControl0

8位读/写DMACCxControl0寄存器包含DMA通道控制信息，如脉冲大小和传输宽度。通道启动前，每个寄存器都直接通过软件编程。当DMA通道启动时，随着目标地址的增加，该寄存器不断进行更新。当完成一个数据包的传输时，读这个寄存器，同时通道有效，但不发送有用的信息。这是因为通过时间，软件处理读取值，通道可能有进步。当通道停止时，只能故意被读取。表4-47所示为DMACCxControl0寄存器的位分配。

表4-47 DMACCxControl0寄存器的位分配

| DMACCxControl0 | 位 | 读/写 | 功 能 |
|---|---|---|---|
| I | [31] | 读/写 | 终端计数中断启动位。控制当前的LLI是否期望引发终端计数中断 |
| Prot | [30:28] | 读/写 | 保护 |
| DI | [27] | 读/写 | 目标增量。每个传输后，随设置的目标地址递增 |
| SI | [26] | 读/写 | 源增量。每个传输后，随设置的源地址递增 |
| D | [25] | 读/写 | 目标AHB主机选择：<br>0 = AHB主机1（AXI_SPINE），用于目标传输选择<br>1 = AHB主机2（AXI_PERI），用于目标传输选择 |
| S | [24] | 读/写 | 源AHB主选择：<br>0 = AHB主机1（AXI_SPINE），用于源传输选择<br>1 = AHB主机2（AXI_PERI），用于源传输选择 |
| Dwidth | [23:21] | 读/写 | 目标传输宽度<br>传输宽度比AHB主总线宽度宽是非法的。源与目标宽度和其他的宽度是不同的。硬件自动压缩和解压数据包作为请求 |
| Swidth | [20:18] | 读/写 | 源传输宽度<br>传输宽度比AHB主总线宽度宽是非法的。源与目标宽度和其他的宽度是不同的。硬件自动压缩和解压数据包作为请求 |
| DBSize | [17:15] | 读/写 | 目标脉冲大小<br>显示传输的数目，提出一个目标脉冲传输请求 。这个值必须设置为目标外设的脉冲大小，或如果目标是内存，就是内存边界大小。脉冲大小是数据量，当DMACxBREQ信号在目标外设中有效时，脉冲大小被传输。脉冲大小与AHB HBURST没有关系 |

（续）

| DMACCxControl0 | 位 | 读/写 | 功　能 |
| --- | --- | --- | --- |
| SBSize | [14:12] | 读/写 | 源脉冲大小<br>显示传输数目，提出一个源脉冲。这个值必须设置为源外设的脉冲大小，或如果源是内存，就是内存边界大小。脉冲大小是数据量，当DMACxBREQ信号在源外设中有效时，脉冲大小被传输。脉冲大小与AHB HBURST没有关系 |
| Reserved | [11:0] | 读 | 保留 |

### 7. 通道控制寄存器DMACCxControl1

8位读/写DMACCxControl1寄存器，它包含DMA通道控制信息，如脉冲大小和传输宽度。在DMA通道启动之前，每个寄存器通过软件直接编程。当DMA通道启动时，随着目标地址的增加，该寄存器不断更新。当通道有效，但不提供有用的信息时，读这个寄存器。这是因为通过时间，软件处理读取值，通道可能有进步。当通道停止时，只能故意被读取，在这样情况下，它显示最后信息，源地址被读取。表4-48所示为DMACCxControl1寄存器的位分配。

表4-48 DMACCxControl1寄存器的位分配

| DMACCxControl | 位 | 读/写 | 功　能 |
| --- | --- | --- | --- |
| TransferSize | [24:0] | 读/写 | 传输大小。用于写入，当DMA控制器是流量控制器时，此栏显示传输（源的宽度）的数目来执行。用于读取，传输大小在目标总线上显示传输完成的数目。当通道有效但没有给予有意义的值时，软件已经处理该读取的值，通道已经进行，这时读该寄存器<br>当通道启动和禁止时，特意被使用。如果DMAC控制器不是流量控制器，传输值的大小不可以被使用 |

### 8. 通道配置寄存器DMACCxConfiguration

8位读/写DMACCxConfiguration寄存器，用于配置DMA通道。当请求一个新的LLI时，该寄存器不更新。表4-49所示为DMACCxConfiguration寄存器的位的分配情况。

表4-49 DMACCxConfiguration寄存器的位分配

| DMACCxConfiguration | 位 | 读/写 | 功　能 |
| --- | --- | --- | --- |
| H | [18] | 读/写 | 停止：<br>0 = 允许DMA请求<br>1 = 屏蔽进一步的源DMA请求<br>FIFO通道的内容被传输完成。该值有效，并通道启动位完全禁止一个DMA通道 |
| A | [17] | 读 | 有效：<br>0 = 通道的FIFO中没有数据<br>1 = 通道的FIFO中有数据<br>该值有效，并通道启动位完全禁止DMA通道 |

（续）

| DMACCxConfiguration | 位 | 读/写 | 功　　能 |
|---|---|---|---|
| L | [16] | 读/写 | 锁<br>设置该位来使锁定的传输启动 |
| ITC | [15] | 读/写 | 终端计数中断屏蔽。当清除该位时，屏蔽相关通道的终端计数错误中断 |
| IE | [14] | 读/写 | 中断错误屏蔽。当清除该位时，屏蔽相关通道的错误中断 |
| FlowCntrl | [13:11] | 读/写 | 流控制和传输类型。该值用于指示流控制器和传输类型。支持的流控制器只有DMA控制器。传输类型有存储器到存储器、存储器到外设、外设到存储器、或者外设到外设 |
| Reserved（OneNandModeDst） | [10] | 读/写 | 保留，必须写入0，并且屏蔽读取操作<br>用于DMAC0和SDMAC0的通道3，该位用于支持页写入类型。如果该位设置为1，并且目标地址指向OneNAND控制器的地址域，则目标地址增量设置为支持OneNAND控制器的01指令。当该位设置为1，D应当是AHB主控器1，D1应当是增量，DWidth应当为字，DBSize应当是4的倍数 |
| DestPeripheral | [9:6] | 读/写 | 目标外设。该值选择DMA目标请求外设。如果是传输到存储器，则该区域屏蔽 |
| Reserved（OneNandModeSrc） | [5] | 读/写 | 保留。必须写0，并且屏蔽读取<br>对于通道3，该位用于支持页写入类型。如果该位设置为1，并且目标地址指向OneNAND控制器的地址域，则目标地址增量设置支持OneNAND控制器的01指令。当该位设置为1，D应当是AHB主控器1，D1应当是增量，DWidth应当为字，DBSize应当是4的倍数 |
| SrcPeripheral | [4:1] | 读/写 | 源外设。该值选择DMA源请求外设。如果传输源是存储器，则屏蔽该区域 |
| E | [0] | 读/写 | 通道有效。读取该位来指示当前通道是有效还是无效<br>0＝通道无效<br>1＝通道有效<br>通过读取DMACEnbldChns寄存器也能知道通道有效位的状态<br>通过设置该位来使通道无效。当前AHB传输完成并且通道无效时，通道FIFO的数据将丢失。通过设置通道有效位来重启通道。有不可预知的结果，并且通道必须充分初始化<br>当最后的LLI被读取，或者有通道错误，通道有效位被清除，通道也无效<br>如果通道不得不停止，并且不丢失通道FIFO的数据，则必须设置停止位以屏蔽进一步的DMA请求。必须轮询激活位直到它为0，也就是FIFO中没有剩余数据了。最后可以清除通道有效位了 |

### 4.4.3 DMA控制器应用举例

下面是ARM11处理器中，内存到内存传输处理的具体代码实现：

```
static void DMAT_MemtoMem(void)      //功能函数
{
    u32 i,csel,usel;
    u32 uTsize,uBurst,uCh;
// DMA控制器的4种方式选择
Disp("\nSelect DMA Controller : 0:DMAC0,1:DMAC1,2:SDMAC0,3:SDMAC1: ");
    csel=GetIntNum();
    Disp("\n");
    switch(csel)
    {
    case 0:                                              //DMAC0选择
        Disp("Selected DMAC 0 ..... \n");
        DMAC_InitCh(DMA0,DMA_ALL, &oDmac0);
        INTC_SetVectAddr(NUM_DMA0,   Dma0Done);
        INTC_Enable(NUM_DMA0);
    break;
    case 1:                                                //DMAC1选择
        Disp("Selected DMAC 1 ..... \n");
        DMAC_InitCh(DMA1,DMA_ALL, &oDmac1);
        INTC_SetVectAddr(NUM_DMA1,   Dma1Done);
        INTC_Enable(NUM_DMA1);
    break;
    case 2:                                                //SDMAC0选择
        Disp("Selected SDMAC 0 ..... \n");
        DMAC_InitCh(SDMA0,DMA_ALL, &oDmac2);
        INTC_SetVectAddr(NUM_SDMA0,Sdma0Done);
        INTC_Enable(NUM_SDMA0);
    break;
    case 3:                                                //SDMAC1选择
        Disp("Selected SDMAC 1 ..... \n");
        DMAC_InitCh(SDMA1,DMA_ALL, &oDmac3);
        INTC_SetVectAddr(NUM_SDMA1,Sdma1Done);
        INTC_Enable(NUM_SDMA1);
    break;
    default : Assert(0);
    }
//通道选择
Disp("\nSelect Channel : 0:CH0,1:CH1,2:CH2,3:CH3,4:CH4,5:CH5,6:CH6,7:CH7: ");
    usel=GetIntNum();
    Disp("\n");
switch(usel)
    {
    }

    case 0:
        uCh= DMA_A;
    break;
```

```
        case 1:
            uCh= DMA_B;
        break;
        case 2:
            uCh= DMA_C;
        break;
        case 3:
            uCh = DMA_D;
        break;
        case 4:
            uCh = DMA_E;
        break;
        case 5:
            uCh = DMA_F;
        break;
        case 6:
            uCh = DMA_G;
        break;
        case 7:
            uCh = DMA_H;
        break;
        default : Assert(0);
    //传输宽度选择
    Disp("\nSelect Transfer Width : 0:BYTE,1:HWORD,2:WORD:");
        usel=GetIntNum();
        Disp("\n");
        switch(usel)
        {
        case 0:                                              //字节选择
            uTsize = BYTE;
        break;
        case 1:                                              //半字选择
            uTsize = HWORD;
        break;
        case 2:                                              //字选择
            uTsize = WORD;
        break;
        default : Assert(0);
        }
    //脉冲大小选择
    Disp("\nSelect Burst Size : 0:SINGLE,1:BURST4,2:BURST8,3:BURST16,4:BURST32,5:BURS
T64,6:BURST128,7:BURST256:");
        usel=GetIntNum();
        Disp("\n");
        switch(usel)
        {
        case 0:
            uBurst = SINGLE;
        break;
        case 1:
```

```
        uBurst = BURST4;
    break;
    case 2:
        uBurst = BURST8;
    break;
    case 3:
        uBurst = BURST16;
    break;
    case 4:
        uBurst = BURST32;
    break;
    case 5:
        uBurst = BURST64;
    break;
    case 6:
        uBurst = BURST128;
    break;
    case 7:
        uBurst = BURST256;
    break;
    default : Assert(0);
    }
//传输大小选择
Disp("\nSelect Transfer Size [1~0x200_0000] :  ");
    uDataCnts=GetIntNum();
    Disp("\n");
//0.清除rx/tx缓冲器
    for(i = 0; i<(uDataCnts*uTsize+16); i++)
    {
    *(u8 *)(uRxBuffAddr+i) = 0;

    *(u8 *)(uTxBuffAddr+i) = 0;
    }
//1.建立tx缓冲器
    for(i = 0; i<uDataCnts*uTsize; i++)
    *(u8 *)(uTxBuffAddr+i) = (u8)(i+2)%0xff;
    switch(csel)
    {
    case 0 :
//通道0,链表地址,源地址,源类型,目标地址,目标类型,传输宽度,传输大小
    OpMode(DEMAND),Src Req,Dst Req,Burst
    DMACH_Setup((DMA_CH)uCh,0x0,uTxBuffAddr,0,uRxBuffAddr,0, (DATA_SIZE)uTsize,
    uDataCnts,DEMAND,MEM,MEM, (BURST_MODE)uBurst, &oDmac0);
    DMACH_Start(&oDmac0);
    break;
    case 1 :
//通道1,链表地址,源地址,源类型,目标地址,目标类型,传输宽度,传输大小
    OpMode(DEMAND),Src Req,Dst Req,Burst
    DMACH_Setup((DMA_CH)uCh,0x0,uTxBuffAddr,0,uRxBuffAddr,0, (DATA_SIZE)uTsize,
```

```
        uDataCnts,DEMAND,MEM,MEM, (BURST_MODE)uBurst, &oDmac1);
        DMACH_Start(&oDmac1);
        break;
        case 2 :
    //通道2,链表地址,源地址,源类型,目标地址,目标类型,传输宽度,传输大小
        OpMode(DEMAND),Src Req,Dst Req,Burst
        DMACH_Setup((DMA_CH)uCh,0x0,uTxBuffAddr,0,uRxBuffAddr,0, (DATA_SIZE)uTsize,
        uDataCnts,DEMAND,MEM,MEM, (BURST_MODE)uBurst, &oDmac2);
        DMACH_Start(&oDmac2);
        break;
        case 3 :
    //通道3,链表地址,源地址,源类型,目标地址,目标类型,传输宽度,传输大小
        OpMode(DEMAND),Src Req,Dst Req,Burst
        DMACH_Setup((DMA_CH)uCh,0x0,uTxBuffAddr,0,uRxBuffAddr,0, (DATA_SIZE)uTsize,
        uDataCnts,DEMAND,MEM,MEM, (BURST_MODE)uBurst, &oDmac3);
        DMACH_Start(&oDmac3);
        break;
        default :
        Assert(0);
    }
    while(g_DmaDone==0);
    if(CompareDMA(uTxBuffAddr,uRxBuffAddr, (DATA_SIZE)uTsize,uDataCnts))
    Disp(" >> Test Tx&Rx -> Ok << \n");
    else
        {
        Disp(">>*** Tx-data & Rx-data mismatch ***<< \n");
        }

    switch(csel)
        {
        case 0:
            INTC_Disable(NUM_DMA0);
            DMAC_Close(DMA0,DMA_ALL, &oDmac0);
        break;
        case 1:
            INTC_Disable(NUM_DMA1);
        }
            DMAC_Close(DMA1,DMA_ALL, &oDmac1);
        break;
        case 2:
            INTC_Disable(NUM_SDMA0);
            DMAC_Close(SDMA0,DMA_ALL, &oDmac2);
        break;
        case 3:
            INTC_Disable(NUM_SDMA1);
            DMAC_Close(SDMA1,DMA_ALL, &oDmac3);
        break;
        default : Assert(0);
    }
```

# 4.5　UART串行接口

## 4.5.1　UART串行接口简介

S3C6410提供了4个独立的异步串行口，每个独立的串行口可以工作在中断模式或DMA模式下。也就是说UART能够产生一个中断或DMA请求。UART可以提供的速率最高为3Mbps。每个UART包含2个64位的FIFO，分别用于接收和发送缓冲。

每个UART包含一个波特率发生器、发送器、接收器和控制单元。该波特率发生器由PCLK、EXT_UCLK0或EX_UCLK1进行时钟控制。发射器和接收器包含64字节的FIFO存储器和数据移位寄存器。发送数据之前，首先将数据写入FIFO存储器，然后复制到发送移位寄存器。通过发送数据的引脚TxDn将数据发送，同时，通过数据接收的引脚RxDn将接收到的数据从接收移位寄存器复制到FIFO存储器。UART的结构框图如图4-11所示。

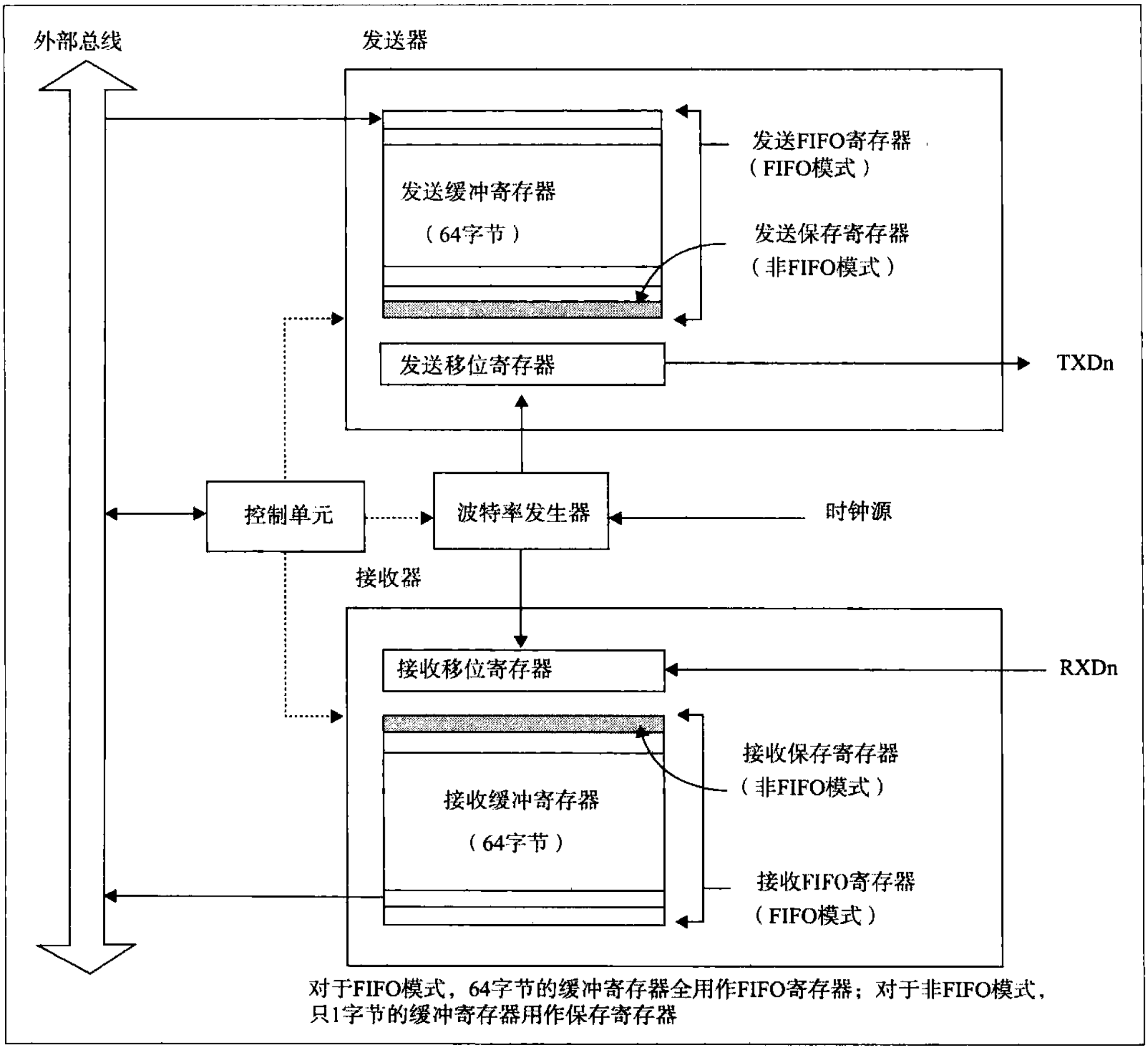



图4-11　UART结构框图

## 4.5.2 UART的操作

下面介绍UART的操作，包括数据发送、数据接收、中断产生和自动流量控制。

**1. 数据发送**

数据（帧）发送是可编程的。数据帧是由一个起始位、5～8个数据位、一个可选的奇偶位和1～2个可由行控制寄存器（ULCONn）指定的停止位组成。发送器也可以产生中断条件，在传输过程中，它通过置位逻辑状态0来强制串行输出。当目前的发送完全传输完成后，发送中断信号。然后不断传送数据到发送FIFO寄存器（在非FIFO的模式下，发送到保存寄存器）。

**2. 数据接收**

数据接收和数据发送一样，数据帧接收也是可编程的。数据帖是由一个起始位、5～8个数据位、一个可选的奇偶位和行控制寄存器指定的1～2个停止位组成。接收器可以检测到溢出错误、奇偶错误、帧错误和中断条件，并为它们设置错误标志。溢出错误说明在数据被读取之前，新的数据已经将原有的数据覆盖。奇偶错误说明接收器已经检测到一个意外的奇偶条件。帧错误表示收到的数据没有有效的停止位。中断条件表明接收过程中置位逻辑状态0的时间比发送一帧的时间长。当三个字的时间（间隔由设置的字长决定）间隔内没有接收任何数据，并且FIFO模式下接收FIFO寄存器不为空，接收超时条件发生。

**3. 自动流量控制（AFC）**

该S3C6410的UART0和UART1通过nRTS和nCTS信号支持自动流量控制。某种情况下，它可以连接到外部UART。如果想要将UART和一台调制解调器连接，必须在UMCONn寄存器中禁用自动流量控制位，并且通过软件控制信号nRTS。在自动流量控制过程中，nRTS依靠接收器的条件，通过nCTS信号控制发送器的运作。只有当nCTS信号被激活（在AFC中，nCTS意味着另一个UART的FIFO寄存器准备接收数据）时，UART的发送器才发送FIFO寄存器中的数据。在UART接收数据之前，如果它接收FIFO寄存器超过2字节以上的空间，则nRTS被激活；如果它的接收FIFO寄存器只有不足1字节的空间，则nRTS停止活动（在AFC中，nRTS意味着它本身的接收FIFO寄存器已经被接收数据）。图4-12所示为UARTAFC接口的发送和接收状态图。

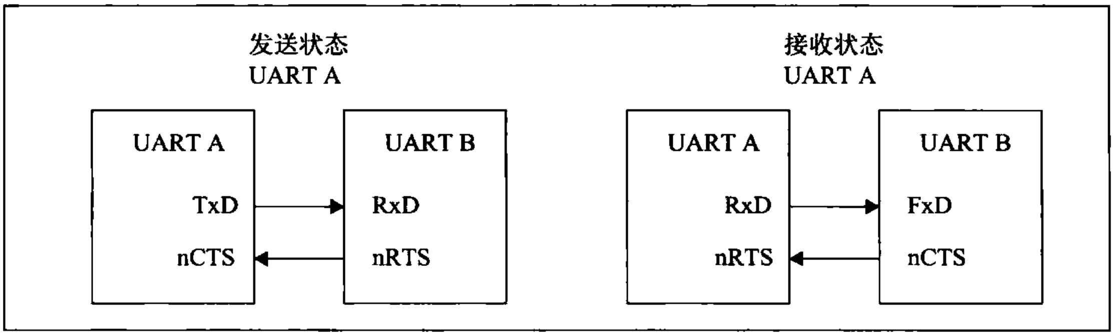



图4-12 UARTAFC接口的发送和接收状态图

#### 4. RS-232C接口

要将UART连接到调制解调器接口（而不是零调制解调器），需要nRTS、nCTS、nDSR、nDTR、DCD和nRI信号。在这种情况下，可以通过软件来控制这些信号与一般的I/O端口。

#### 5. 中断/DMA请求的产生

每个S3C6410的UART有7个状态（发送/接收/错误）信号：溢出错误、奇偶错误、帧错误、中断、接收缓冲区数据就绪、传输缓冲区为空和发送移位寄存器为空。其状态信号靠相应的UART的状态寄存器（UTRSTATn/UERSTATn）来指示。溢出错误、奇偶错误、帧错误和中断是指由于收到错误的信息，每一种都可以引起错误接收错误状态中断请求，如果在控制寄存器UCONn中将接收错误状态中断使能位设置为1，当检测到一个接收错误状态中断请求，可通过读UERSTSTn的值来辨别信号。当接收器将数据从接收移位寄存器传送到接收FIFO寄存器（在FIFO模式下），并且数量达到RXFIFO触发电平时，则接收中断产生。如果控制寄存器（UCONn）中接收模式被设置为1（中断请求或轮询模式），则接收中断产生。非FIFO模式中，在中断请求和轮询模式下，数据从接收移位寄存器传输到接收保存寄存器时会引发接收中断。

当发送器将数据从发送FIFO寄存器传输到它的发送移位寄存器，并且发送FIFO剩余的数据数量达到TXFIFO触发水平时，产生发送中断。如果控制器的传输模式选定为中断请求或轮询模式，则产生发送中断。

非FIFO模式中，在中断请求和轮询模式下，数据从发送保存寄存器传输到发送移位寄存器会引发发送中断。

### 4.5.3 UART专用寄存器

#### 1. UART行控制寄存器

UART模块包括4个行控制寄存器，即ULCON0、ULCON1、ULCON2和ULCON3。行控制寄存器的位定义如表4-50、表4-51所示。

表4-50 UART行控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| ULCON0 | 0x7F005000 | 读/写 | UART 0通道行控制寄存器 | 0x00 |
| ULCON1 | 0x7F005400 | 读/写 | UART1通道行控制寄存器 | 0x00 |
| ULCON2 | 0x7F005800 | 读/写 | UART2通道行控制寄存器 | 0x00 |
| ULCON3 | 0x7F005C00 | 读/写 | UART3通道行控制寄存器 | 0x00 |

表4-51 UART行控制寄存器的位分配

| 位名称 | 位 | 描述 | 初始状态 |
|---|---|---|---|
| Reserved | [7] | 保留 | 0 |

（续）

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Infra-Red Mode | [6] | 确定是否采用红外模式：<br>0 = 普通操作模式<br>1= 红外线输出/接收模式 | 0 |
| Parity Mode | [5:3] | 确定奇偶产生类型和校验，在UART发送/接收操作过程中<br>0xx = 无校验<br>100 =奇校验<br>101 = 偶校验<br>110 = 奇偶强制/校验为1<br>111 = 奇偶强制/校验为0<br>111 = 强制为0 | 000 |
| Number of Stop Bit | [2] | 确定每帧中停止位个数：<br>0 = 每帧1位停止位<br>1 = 每帧2位停止位 | 0 |
| Word Length | [1:0] | 确定每帧中数据位的个数：<br>00 = 5位 01 = 6位<br>10 = 7位 11 = 8位 | 00 |

## 2. UART控制寄存器

UART控制寄存器也有4个，即UCON0、UCON1、UCON2和ULCON3，如表4-52、表4-53所示。

表4-52 UART控制寄存器

| 寄存器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| UCON0 | 0x7F005004 | 读/写 | UART 0通道控制器 | 0x00 |
| UCON1 | 0x7F005404 | 读/写 | UART1通道控制器 | 0x00 |
| UCON2 | 0x7F005804 | 读/写 | UART2通道控制器 | 0x00 |
| UCON3 | 0x7F005C04 | 读/写 | UART3通道控制器 | 0x00 |

表4-53 UART控制寄存器的位分配

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Clock Selection | [11:10] | 选择PCLK或者EXT_UCLK0$_4$）作为UART波特率时钟<br>x0=PCLK:DIV_VAL =（PCLK /（b/s × 16））-1<br>01=EXT_UCLK0:DIV_VAL =（EXT_UCLK /（b/s × 16））-1<br>11=EXT_UCLK1:DIV_VAL=（EXT_UCLK /（b/s × 16））-1 | 0 |
| Tx中断类型 | [9] | 发送中断请求类型<br>0 = 脉冲（当非FIFO模式下的发送缓冲器为空或者FIFO模式下发送FIFO达到触发脉冲时产生中断）<br>1 = 电平（当非FIFO模式下的发送缓冲为空或者FIFO模式下发送FIFO达到触发电平时产生中断） | 0 |

（续）

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Rx中断类型 | [8] | 接收中断请求类型<br>0 = 脉冲（在非FIFO模式下接收缓冲区接收到数据或者在FIFO模式下达到接收FIFO触发脉冲时，请求中断）<br>1 = 电平 （当非FIFO模式下接收缓冲区接收到数据或者在FIFO模式下达到接收FIFO触发电平时，请求中断 | 0 |
| Rx Time Out Enable | [7] | 使能/禁止接收超时中断，当UART FIFO使能时<br>0 = 禁止 1 = 使能 | 0 |
| Rx Error Status InterruptEnable | [6] | 在接收过程中，如果发生帧错误或溢出错误，使能/禁止UART产生中断<br>0 = 不产生接收错误状态中断<br>1 = 产生接收错误状态中断 | 0 |
| Loop-back Mode | [5] | 设置环回位为1，使UART进入环回模式。此模式仅为测试目的使用<br>0 = 普通操作 1 = 环回模式 | 0 |
| Send Break Signal | [4] | 置环回位为1使UART进入环回模式。这种模式只为测试提供参考<br>0=正常发送 1= 发送中断信号 | 0 |
| Transmit Mode | [3:2] | 确定哪个模式可以写发送数据到UART发送缓冲寄存器<br>00= 禁止<br>01= 中断请求或轮询模式<br>10=DMA0请求（仅用于UART0），DMA3请求（请求信号0）<br>11= DMA1请求（仅请求信号1） | 00 |
| Receive Mode | [1:0] | 确定哪个模式可以从UART接收缓冲寄存器读数据<br>00 =禁止<br>01 = 中断请求或轮询模式<br>10=DMA0请求（仅用于UART0），DMA3请求（请求信号0）<br>11= DMA1请求（请求信号1） | 00 |

### 3. UART的FIFO控制寄存器

UART模块中含有4个UART FIFO控制寄存器，如表4-54、表4-55所示。

表4-54 UART FIFO控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| UFCON0 | 0x7F005008 | 读/写 | UART0通道FIFO控制寄存器 | x00 |
| UFCON1 | 0x7F005408 | 读/写 | U ART1通道FIFO控制寄存器 | x00 |
| UFCON2 | 0x7F005808 | 读/写 | U ART2通道FIFO控制寄存器 | x00 |
| UFCON3 | 0x7F005C08 | 读/写 | U ART3通道FIFO控制寄存器 | x00 |

表4-55 UART FIFO控制寄存器的位分配

| 位名称 | 位 | 描述 | 初始状态 |
|---|---|---|---|
| Tx FIFO Trigger Level | [7:6] | 确定发送FIFO的触发条件<br>00 = 空 01 = 4字节<br>10 = 8字节 11 = 12字节 | 00 |
| Rx FIFO Trigger Level | [5:4] | 确定接收FIFO的触发条件<br>00 = 4字节 01 = 8字节<br>10 = 12字节 11 = 16字节 | 00 |
| Reserved | [3] | 保留 | 0 |
| Tx FIFO Reset | [2] | Tx复位，该位在FIFO复位后自动清除<br>0 = 正常 1= Tx FIFO复位 | 0 |
| Rx FIFO Reset | [1] | Rx复位，该位在FIFO复位后自动清除<br>0 = 正常 1= Rx FIFO复位 | 0 |
| FIFO Enable | [0] | 0 =FIFO禁止 1 = FIFO模式 | 0 |

### 4. UART Modem控制寄存器

UART模块中有两个UART MODEM控制寄存器UMCON0和UMCON1，如表4-56、表4-57所示。

表4-56 UART MODEM控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| UMCON0 | 0x7F00500C | 读/写 | U ART0通道Modem控制寄存器 | 0x0 |
| UMCON1 | 0x7F00540C | 读/写 | U ART1通道Modem控制寄存器 | 0x0 |
| Reserved 0 | x7F00580C | — | 保留 | 未定义 |
| Reserved 0 | x7F005C0C | — | 保留 | 未定义 |

表4-57 UART MODEM控制寄存器的位分配

| 位名称 | 位 | 描述 | 初始状态 |
|---|---|---|---|
| Reserved | [7:5] | 当AFC被激活，这个位决定什么时候阻止信号<br>000 =接收FIFO控制63字节<br>001 =接收FIFO控制56字节<br>010 =接收FIFO控制48字节<br>011 =接收FIFO控制40字节<br>100 =接收FIFO控制32字节<br>101 =接收FIFO控制24字节 | 000 |
| Auto Flow Control (AFC) | [4] | AFC是否允许：<br>0 = 禁止 1 =激活 | 0 |
| Reserved | [3:1] | 这3位必须均为0 | 00 |

（续）

| 位　名　称 | 位 | 描　　述 | 初始状态 |
|---|---|---|---|
| Request to Send | [0] | 如果AFC位允许，则该位忽略，这时，S3C6410将自动控制nRTS<br>如果AFC位禁止，则nRTS必须被软件控制<br>0 = 'H' 电平（nRTS无效）<br>1 = 'L' 电平（nRTS有效） | 0 |

### 5. UART接收（Rx）/发送（Tx）状态寄存器

UART模块有4个UART接收/发送状态寄存器，即UTRSTAT0、UTRSTAT1、UTRSTAT2和UTRSTAT3，如表4-58、表4-59所示。

表4-58　UART接收/发送状态寄存器

| 寄　存　器 | 地　　址 | 读/写 | 描　　述 | 复位值 |
|---|---|---|---|---|
| UTRSTAT0 | 0x7F005010 | 读 | UART 0通道Tx/Rx状态寄存器 | 0x6 |
| UTRSTAT1 | 0x7F005410 | 读 | UART 1通道Tx/Rx状态寄存器 | 0x6 |
| UTRSTAT2 | 0x7F005810 | 读 | UART 2通道Tx/Rx状态寄存器 | 0x6 |
| UTRSTAT3 | 0x7F005C10 | 读 | UART 3通道Tx/Rx状态寄存器 | 0x6 |

表4-59　UART接收/发送状态寄存器的位分配

| 位　名　称 | 位 | 描　　述 | 初始状态 |
|---|---|---|---|
| Transmitter empty | [2] | 在发送缓冲寄存器没有有效数据或发送移位寄存器为空时，该位自动置1<br>0 = 不空<br>1 = 发送器（发送缓冲寄存器和移位寄存器）空 | 1 |
| Transmit buffer empty | [1] | 当发送缓冲寄存器为空时，该位自动置1<br>0 =发送缓冲寄存器不空<br>1 =空<br>（在非FIFO模式，中断或DMA被申请。在FIFO模式，当Tx FIFO触发电平被设置为00（空）时，中断或DMA被申请。）<br>如果UART使用FIFO，则用户应该检查UFSTAT寄存器的Tx FIFO计数位和Tx FIFO满标志位，以代替检查该位 | 1 |
| Receive data ready buffer | [0] | 无论何时接收缓冲寄存器包含在RXDn接口接收的有效数据，该位自动置1<br>0 = 空<br>1 = 接收缓冲寄存器有接收数据（在非FIFO模式下，中断或DMA被申请）<br>如果UART使用FIFO，则用户应该检查UFSTAT寄存器中的Rx FIFO计数位和Rx FIFO满标志位以代替检查该位 | 0 |

### 6. UART发送缓冲寄存器（保存寄存器和FIFO寄存器）

UART模块有4个发送缓冲寄存器寄存器，即UTXH0、UTXH1、UTXH2和UTXH3。

UTXHn有一个8位数据作为发送数据，如表4-60所示。

表4-60 UART发送缓冲寄存器

| 寄存器 | 地址 | R/写 | 描述 | 复位值 |
|---|---|---|---|---|
| UTXH0 | 0x7F005020 | 写 | UART 0通道发送缓冲寄存器 | — |
| UTXH1 | 0x7F005420 | 写 | UART 1通道发送缓冲寄存器 | — |
| UTXH2 | 0x7F005820 | 写 | UART 2通道发送缓冲寄存器 | — |
| UTXH3 | 0x7F005C20 | 写 | UART 3通道发送缓冲寄存器 | — |

### 7. UART接收缓冲寄存器（保存寄存器和FIFO寄存器）

UART模块有四个接收缓冲寄存器寄存器：URXH0、URXH1、URXH2和URXH3。URXHn有一个8位数据作为发送数据，如表4-61所示。

表4-61 UART接收缓冲寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| URXH0 | 0x7F005024 | 读 | UART 0通道接收缓冲寄存器 | — |
| URXH1 | 0x7F005424 | 读 | UART 1通道接收缓冲寄存器 | — |
| URXH2 | 0x7F005824 | 读 | UART 2通道接收缓冲寄存器 | — |
| URXH3 | 0x7F005C24 | 读 | UART 3通道接收缓冲寄存器 | — |

### 8. UART波特率分频寄存器

UART模块有4个波特率分频寄存器，即UBRDIV0、UBRDIV1、UBRDIV2和UBRDIV3。UBRDIVn中的值决定串行Tx/Rx时钟波特率，具体如下：

```
DIV_VAL = UBRDIVn + (UDIVSLOTn中1的量)/16
DIV_VAL = (PCLK / (b/s×16)) -1
DIV_VAL = (EXT_UCLK0 / (b/s×16) ) -1
```

或者

```
DIV_VAL = (EXT_UCLK1 / (b/s×16) ) -1
```

除数的范围为1 ~（$2^{16}$−1），并且UEXTCLK应该比PCLK小。

利用UDIVSLOT能够得到更准确的波特率。例如，如果波特率是115200 bps PCLK、EXT_UCLK0或EXT_UCLK1是40 MHz，则UBRDIVn和UDIVSLOTn是：

```
DIV_VAL = (40000000 / (115200×16) ) -1
        = 21.7 -1
        = 20.7
UBRDIVn = 20 (DIV_VAL的整数部分)
(UDIVSLOTn中1的数量) /16 = 0.7
```

这时，（UDIVSLOTn中1的数量）= 11。因此，UDIVSLOTn为16'b1110_1110_1110_1010或16'b0111_0111_0111_0101等。UDIVSLOTn选择如表4-62所示。

表4-62　UDIVSLOTn的选择

| Num of 1's | UDIVSLOTn | Num of 1's | UDIVSLOTn |
|---|---|---|---|
| 0 | 0x0000（0000_0000_0000_0000b） | 8 | 0x5555（0101_0101_0101_0101b） |
| 1 | 0x0080（0000_0000_0000_1000b） | 9 | 0xD555（1101_0101_0101_0101b） |
| 2 | 0x0808（0000_1000_0000_1000b） | 10 | 0xD5D5（1101_0101_1101_0101b） |
| 3 | 0x0888（0000_1000_1000_1000b） | 11 | 0xDDD5（1101_1101_1101_0101b） |
| 4 | 0x2222（0010_0010_0010_0010b） | 12 | 0xDDDD（1101_1101_1101_1101b） |
| 5 | 0x4924（0100_1001_0010_0100b） | 13 | 0xDFDD（1101_1111_1101_1101b） |
| 6 | 0x4A52（0100_1010_0101_0010b） | 14 | 0xDFDF（1101_1111_1101_1111b） |
| 7 | 0x54AA（0101_0100_1010_1010b） | 15 | 0xFFDF（1111_1111_1101_1111b） |

### 9. UART中断处理寄存器

中断处理寄存器包括产生中断的信息，如表4-63、表4-64所示。

表4-63　UART中断处理寄存器

| 寄存器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| UINTP0 | 0x7F005030 | 读/写 | UART 0通道中断处理寄存器 | 0x0 |
| UINTP1 | 0x7F005430 | 读/写 | UART 1通道中断处理寄存器 | 0x0 |
| UINTP2 | 0x7F005830 | 读/写 | UART 2通道中断处理寄存器 | 0x0 |
| UINTP3 | 0x7F005C30 | 读/写 | UART 3通道中断处理寄存器 | 0x0 |

表4-64　UART中断处理寄存器的位分配

| UINTPn | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| MODEM | [3] | 产生Modem中断 | 0x0 |
| TXD | [2] | 产生发送中断 | 0x0 |
| ERROR | [1] | 产生错误中断 | 0x0 |
| RXD | [0] | 产生接收中断 | 0x0 |

当4位有一位置位逻辑1时，UART每个通道都产生中断，在中断服务程序时这个寄存器被清理。可以通过置1在指定的位来清理UINTP特殊的位。

## 4.5.4 UART接口应用举例

本程序实例实现从UART0接收数据，然后分别从UART0和UART1发送出去。其功能可以把键盘敲击的字符通过PC机的串口发送给ARM系统上的UART0。ARM系统上的UART0接收到字符后，再通过UART0和UART1发送给PC，这样就完成了串口间的收发数据。

### 1. S3C6410的RS-232C接口电路

RS-232C接口电路如图4-13所示。

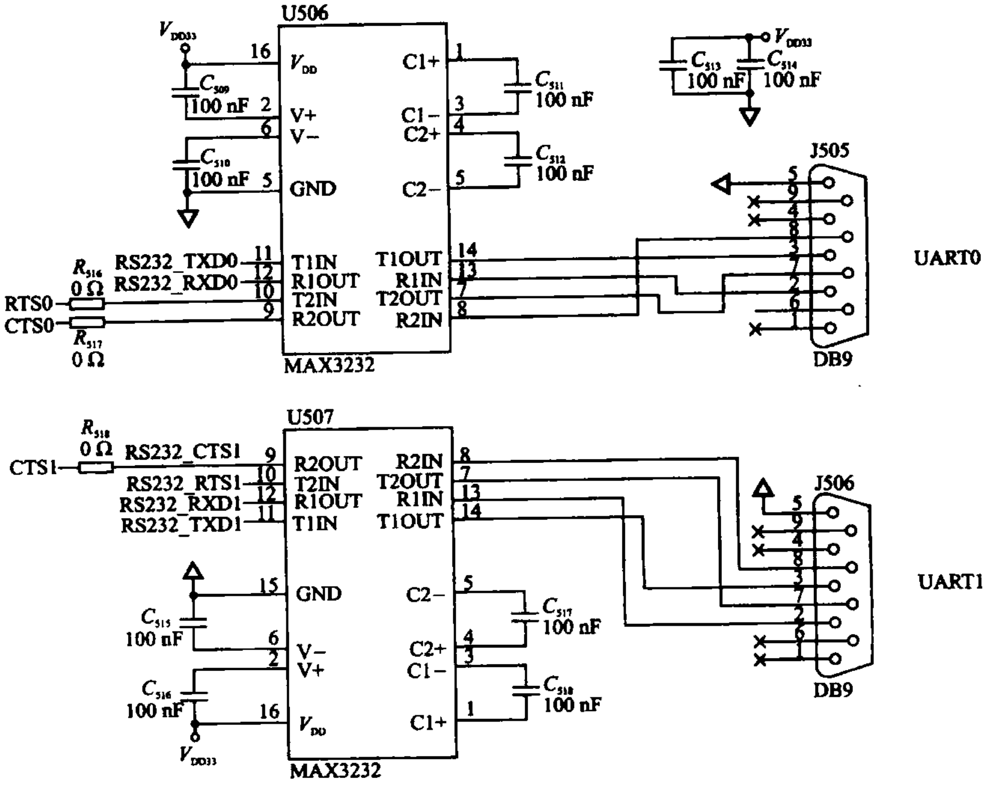



图4-13　S3C6410的RS-232C接口电路

## 2. 串行通信程序

（1）定义与UART相关的寄存器

以UART0为例，需要定义的寄存器如下：

```
#define rULCON0(*(volatile unsigned*)0x7F005000)//UART0行控制寄存器
#define rUCON0(*(volatile unsigned*)0x7F005004) //UART0控制寄存器
#define rUFCON0(*(volatile unsigned*)0x7F005008)//UART0 FIFO控制寄存器
#define rUMCON0(*(volatile unsigned*)0x7F00500C)//UART0 Modem控制寄存器
#define rUTRSTAT0(*(volatile unsigned*)0x7F005010) //UART0 Tx/Rx状态寄存器
#define rUERSTAT0(*(volatile unsigned*)0x7F005014) //UART0 Rx错误状态寄存器
#define rUFSTAT0(*(volatile unsigned*)0x7F005018) //UART0 FIFO状态寄存器
#define rUMSTAT0(*(volatile unsigned*)0x7F00501C)  //UART0 Modem状态寄存器
#define rUBRDIV0(*(volatile unsigned*)0x7F005028)   //UART0波特率系数寄存器
#ifdef__BIG_ENDIAN                                   //大端模式
#define rUTXH0(*(volatile unsigned char*)0x7F005020)
    //UART0发送缓冲寄存器
#define rURXH0(*(volatile unsigned char*)0x7F005024)
    //UART0接收缓冲寄存器
#define WrUTXH0(ch)(*(volatile unsigned char*)0x7F005020)=(unsigned char)(ch)
#define RdURXH0()(*(volatile unsigned char*)0x7F005024)
#define UTXH0(0x7F00501d +3)
   //DMA使用的字节访问地址
#define URXH0(0x50000021+3)
#else//小端模式
#define rUTXH0(*(volatile unsigned char*)0x5000001d)
     //UART0 Transmission Hold
```

```
#define rURXH0(*(volatile unsigned char*)0x50000021)
    //UART0接收缓冲区
#define WrUTXH0(ch)(*(volatile unsigned char*)0x5000001d)=(unsigned char)(ch)
#define RdURXH0()(*(volatile unsigned char*)0x50000021)
#define UTXH0(0x5000001d)
     //DMA使用的字节访问地址
#define URXH0(0x50000021)
#endif
```

（2）初始化操作

参数pclk为时钟源的时钟频率，band为数据传输的波特率，初始化函数Uart_Init( )的实现如下：

```
void Uart_Init(int pclk,int baud) {
          int I;
          if(pclk= =0)
          pclk=PCLK;
          rUFCON0=0x0;              //UARTO FIFO控制寄存器,FIFO禁止
          rUFCON1=0x0;              //UART1 FIFO控制寄存器,FIFO禁止
          rUFCON2=0x0;              //UART2 FIFO控制寄存器,FIFO禁止
          rUMCON0=0x0;              //UARTO MODEM控制寄存器,AFC禁止
          rUMCONI=0x0;              //UART1 MODEM控制寄存器,AFC禁止
          //UART0
          rULCON0=0x3 ;
     //行控制寄存器: 正常模式,无奇偶校验,1位停止位,8位数据位
          rUCON0=0x245 ;  //控制寄存器
          rUBRDIV0 =((int)(pclk/16. /baud+0.5)-1) ;   //波特率因子寄存器
          //UART1
          rULCON1=0x3;
          rUCON1=0x245;
          rUBRDIV1=((int)(pclk/16. /baud)-1);
          //UART2
          rULCON2=0x3;
          rUCON2=0x245;
          rUBRDIV2=((int)(pclk/16. /baud)-1);
          for(i=0; i< 100; i++);
     }
```

（3）发送数据

其中whichUart为全局变量，指示当前选择的UART通道，使用串口发送一个字节的代码如下：

```
void Uart_SendByte(int data) {
        if(whichUart= =0) {
            if(data= ='\n') {
                while(! (rUTRSTAT0&0x2));
                Delay(10);                          //延时,与终端速度有关
                rUTXH0='\r' ;
            }
while(! (rUTRSTAT0&0x2));                           //等待,直到发送状态就绪
            Delay(10);
            rUTXH0=data;
      }

else if(whichUart= =1) {
          if(data=='\n') {
            while(! (rUTRSTAT1&0x2));
            Delay(10);                              //延时,与终端速度有关
            rUTXH1='\r';
```

```
            }
            while(! (rUTRSTAT1&0x2));          //等待,直到发送状态就绪
            Delay(10);
            rUTXH1=data;
        }
    else if(whichUart= =2) {
            if(data= ='\n') {
             while(! (rUTRSTAT2&0x2));
             Delay(10);                        //延时,与终端速度有关
             rUTXH2='\r';
            }
            while(! (rUTRSTAT2&0x2));          //等待,直到发送状态就绪
            Delay(10);
            rUTXH2=data;
        }
    }
```

（4）接收数据

如果没有接收到字符则返回0。使用串口接收一个字符的代码如下：

```
char Uart_GetKey(void) {
        if(whichUart= =0) {
            if(rUTRSTAT0&0x1)                  //UART0接收到数据
                  return RdURXH0();
            else
               return 0;
        }
        else if(whichUart= =1) {
            if(rUTRSTAT1&0x1)                  //UART1接收到数据
                return RdURXH1();
          else
             return 0;
        }
else if(whichUart= =2) {
          if(rUTRSTAT2&0x1)                    //UART2接收到数据
             return RdURXH2();
          else
             return 0;
        } else
          return 0;
    }
```

（5）主函数

主函数实现的功能为从UART0接收字符，然后将接收到的字符分别从UART0和UART1发送出去，其中Uart_Select（n）用于选择使用的传输通道为UARTn。代码如下：

```
#include<string. h>
#include". . \INC\config. h"
void Main(void) {
char data;
Target_Init();
while(1) {
data=Uart GetKey();                     //接收字符
if(data! =0x0) {
Uart_Select(0);.                        //从UART0发送出去
```

```
Uart_Printf("key=%c\n",data);
Uart_Select(1);                          //从UART1发送出去
Uart_Printf("key=%c \n",data);
Uart_Select(0);
        }
    }
}
```

# 4.6　I2C总线接口

## 4.6.1　I2C总线接口简介

S3C6410处理器能支持一个多主控器I2C串行接口。一个专用的串行数据线（SDA）和一个串行时钟线（SCL）在总线主控器和连接到I2C总线的外围设备之间传输数据。SDA和SCL线是双向的。在多主控制I2C总线模式下，多个S3C6410处理器能发送（或接收）串行数据到从属设备。主控器S3C6410能开始和结束I2C总线上的数据传输。在S3C6410中I2C总线使用标准的总线仲裁程序。

为了控制多个I2C总线操作，必须将值写入下面的寄存器：

- 多主控器I2C总线控制寄存器，IICCON；
- 多主控器I2C总线控制/状态寄存器，IICSTAT；
- 多主控器I2C总线发送/接收数据移位寄存器，IICDS；
- 读主控器I2C总线地址寄存器，IICADD。

当I2C总线空闲，SDA和SCL线必须是高电平。SDA从高到低转换能启动一个开始条件。当SCL处于高电平，并保持稳定时，SDA从低位到高位传输能启动一个停止条件。主设备能一直产生开始和停止条件。开始条件产生后，主控器通过在第一次输出的数据字节中写入7位的地址来选择从属器设备。第8位用于确定传输方向（读或写）。到SDA总线上的每一个数据字节总数上必须是8位。在总线传输操作期间，发送或接收字节没有限制，数据一直是先从最高有效位（MSB）发送，并且每个字节后面必须立即跟随确认（ACK）位。I2C总线模块图如图4-14所示。

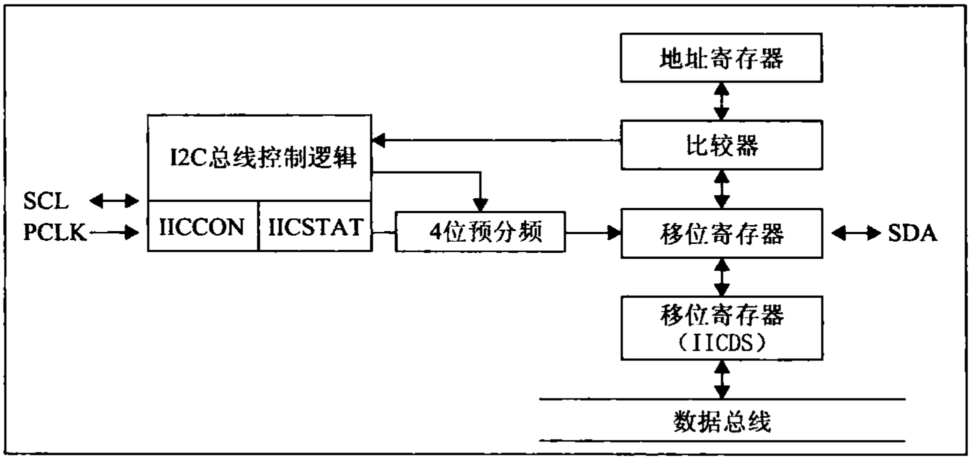



图4-14　I2C总线模块图

## 4.6.2 I2C总线接口操作模式

S3C6410 I2C总线接口有4种操作模式：主控器发送模式、主控器接收模式、从属器发送模式和从属器接收模式。

这些操作模式之间的功能关系描述如下。

### 1. 开始和停止条件

通常在从属器模式下，I2C总线接口无效。在SDA线检测一个开始条件前（当时钟信号SCL在高位时，SDA线发生高位到低位的跃变，开始条件启动），接口必须在从属器模式下。当接口状态变为主控器模式时，在SDA线上的数据传输开始，并且产生SCL信号。

开始条件能通过SDA线传输一个字节的串行数据。一个停止条件能结束该数据传输。主控器能一直产生开始和停止条件。当一个开始条件产生后，I2C总线获得繁忙信号。停止条件将使I2C总线空闲。

当主控器发起一个开始条件时，主控器将发送一个从属地址通知从属设备。一个字节的地址包含7位地址（前7位）和1位传输方向指示（表示写或读，第8位）。如果第8位是0，表示写操作（发送操作）；如果第8位是1，表示请求读取数据（接收操作）。

主控器通过发出一个停止条件来完成传输操作。如果主控器想继续将数据发送到主线，它将产生另一个开始条件和一个从属地址。通过这种方式，读写操作能在不同的格式下被执行。

开始和停止条件模块图如图4-15所示。

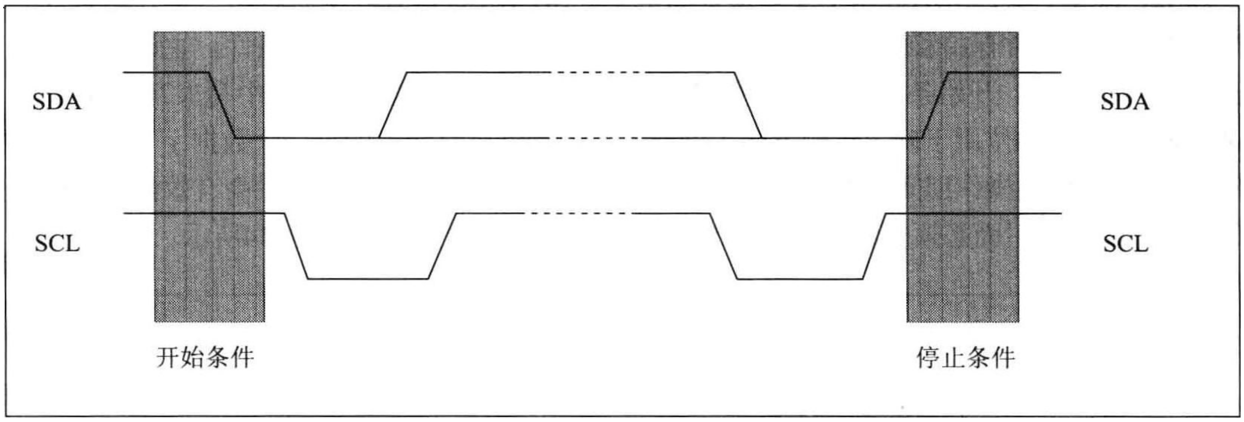



图4-15 开始和停止条件模块图

### 2. 数据传输格式

在SDA线上的每一个字节长度必须是8位。起始条件后的第一个字节有一个地址。当I2C主线在主控器模式下操作时，地址能通过主控器被传输。每一个字节后面跟随一个ACK（acknowledgement）位。MSB位始终首先发送。I2C总线数据传输的模块图如图4-16所示。

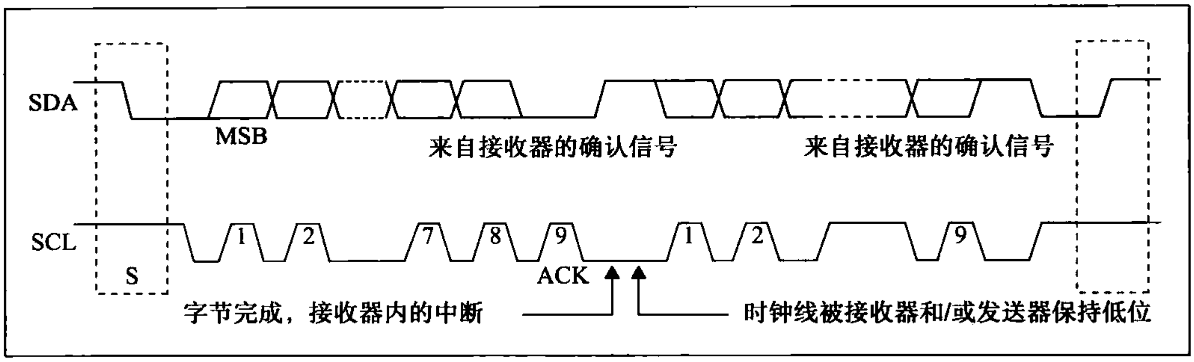



图4-16 I2C总线数据传输的模块图

### 3. ACK信号传输

为了完成一个字节的发送操作，接收器必须将一个ACK位发送到发送器。ACK脉冲在SCL线的第9个时钟时产生。对于发送一个字节来说，8个时钟是必要的。主控器将产生一个时钟脉冲来发送一个ACK位。当ACK时钟脉冲被接收时，通过使SDA置高位，发送器释放SDA线。在传送ACK时钟脉冲期间，接收器驱使SDA线置低位，以使SDA线在第9个SCL脉冲的高位时期保持低位。ACK位传输功能能通过软件（IICSTAT）来激活或者禁止。然而，在SCL的第9个时钟，ACK脉冲被要求来完成一个字节的数据传输操作。

I2C总线上的确认模块图如图4-17所示。

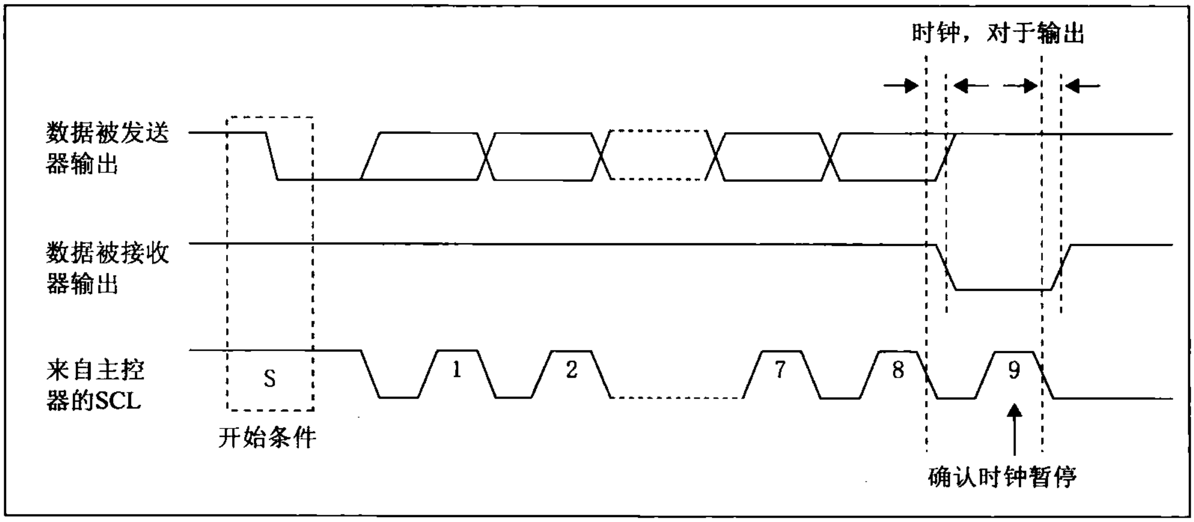



图4-17 I2C总线上的确认模块图

### 4. 读写操作

在发送模式下，当发送数据时，I2C总线接口将一直等待，直到数据移位（I2CDS）寄存器接收到一个新的数据。新的数据写入寄存器之前，SCL线将被保持在低位，数据写入后释放。S3C6410保持中断来确定当前数据发送完成，CPU接收中断请求后，将新的数据写入到IICDS寄存器。

### 5. 异常中断条件

如果一个从属接收器不承认该从属地址，它将保持SDA线为高位。在这种情况下，主控器产生一个中断条件中断传输。中断传输和主控器的接收器有关，来自从属器的最后数据字节被接收后，通过取消一个ACK的产生，通知从属发送器操作结束。从属发送器释放SDA来允许主控器产生一个停止条件。

### 6. I2C总线配置

为了控制串行时钟的频率（SCL），在I2CCON寄存器中，4位的预分频值被执行。I2C总线接口地址被存储在I2C总线地址（I2CADD）寄存器。默认I2C总线地址有一个未知值。

## 4.6.3 I2C总线接口专用寄存器

### 1. 多主控器I2C总线控制（I2CCON）寄存器

多主控器I2C总线控制寄存器及其位分配分别如表4-65、表4-66所示。

表4-65 I2CCON寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| I2CCON | 0x7F004000 | 读/写 | I2C总线控制寄存器 | 0x0X |

表4-66 I2CCON寄存器的位分配

| I2CCON | 位 | 描述 | 初始状态 |
|---|---|---|---|
| 确认产生（1） | [7] | I2C总线确认有效位<br>0：无效<br>1：有效<br>发送模式下，在确认期间，I2CSDA空闲；接收模式下，在确认期间，I2CSDA为L | 0 |
| 发送时钟源选择 | [6] | I2C总线发送时钟预分频选择位的源时钟<br>0: I2CCLK = fPCLK /16<br>1: I2CCLK = fPCLK /512 | 0 |
| 发送/接收中断 （5） | [5] | I2C总线发送/接收中断有效/无效位<br>0：无效 1：有效 | 0 |
| 中断等待标志 （2）（3） | [4] | I2C总线发送/接收中断等待标志。当该位以1被读取时，I2CSDL连接到L并且I2C停止。为了恢复操作，清除该位为0<br>0：无中断等待（读时）<br>清除等待条件并且恢复操作（写时）<br>1：（1）等待中断（读时）<br>（2）N/A（写时） | 0 |

（续）

| I2CCON | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| 发送时钟值 （4） | [3:0] | I2C总线发送时钟预分频<br>I2C总线发送时钟频率由4位预分频值决定，格式如下：<br>发送时钟 = I2CCLK/（I2CCON[3:0]+1） | 未定义 |

### 2. 多主控器I2C总线控制/状态（I2CSTAT）寄存器

多主控器I2C总线控制/状态寄存器及其位分配如表4-67、表4-68所示。

表4-67 I2CSTAT寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| I2CSTAT | 0x7F004004 | 读/写 | I2C总线控制/状态寄存器 | 0x0 |

表4-68 IICSTAT寄存器的位分配

| I2CSTAT | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| 模式选择 | [7:6] | I2C总线主控器/从属器发送/接收模式选择位<br>00：从属器接收模式<br>01：从属器发送模式<br>10：主控器接收模式<br>11：主控器发送模式 | 00 |
| 繁忙信号状态 / START STOP 条件 | [5] | I2C总线繁忙信号状态位<br>0：（读）不繁忙（当读取时）<br>（写）停止信号产生<br>1：（读）繁忙（当读取时）<br>（写）START信号产生<br>在开始信号后，IICDS中的数据将被自动发送 | 0 |
| 连续输出 | [4] | I2C总线数据输出有效/无效位<br>0：无效接收/发送 1：有效接收/发送 | 0 |
| 仲裁状态标志 | [3] | I2C总线裁定程序状态标志位<br>0：总线裁定成功<br>1：在串行I/O过程中，总线裁定失败 | 0 |
| 地址总线作为从属状态标志 | [2] | I2C总线地址作为从属状态标志位<br>0：读I2C总线寄存器后清除<br>1：接收的从属器地址匹配I2CADD中的地址值 | 0 |
| 地址0状态标志 | [1] | I2C总线地址0状态标志位<br>0：当开始/停止条件被检测到时清除<br>1：接收的从属器地址是00000000b | 0 |

（续）

| I2CSTAT | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| 最后接收位状态标志 | [0] | I2C总线最后接收位状态标志位<br>0：最后接收位为0（ACK被接收）<br>1：最后接收位为1（ACK没有被接收） | 0 |

### 3. 多主控器I2C总线地址（I2CADD）寄存器

多主控器I2C总线地址寄存器及其位分配如表4-69、表4-70所示。

表4-69 I2CADD寄存器

| 寄　存　器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| I2CADD | 0x7F004008 | 读/写 | I2C总线地址寄存器 | 0xXX |

表4-70 I2CADD寄存器的位分配

| I2CADD | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| 从属器地址 | [7:0] | 7位从属器地址<br>当I2CSTAT中串行输出有效位=0时，IICADD写有效。不管当前串行输出有效位（I2CSTAT）的设置怎样，I2CADD值都能被读取<br>从属器地址：[7:1]<br>无映射：[0] | XXXXXXXX |

### 4. 多主控制器I2C总线发送/接收数据移位（I2CDS）寄存器

多主控制器I2C总线发送/接收数据移位寄存器及其位分配如表4-71、表4-72所示。

表4-71 IICDS寄存器的位分配

| 寄存器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| I2CDS | 0x7F00400C | 读/写 | I2C总线发送/接收数据移位寄存器 | 0xXX |

表4-72 IICDS寄存器的位分配

| I2CDS | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| 数据移位 | [7:0] | 用于I2C总线发送/接收操作的8位数据移位寄存器<br>当I2CSTAT中串行输出有效位=1，I2CDS写入有效。无论当前串行输出有效位（I2CSTAT）设置怎样，IICDS值都能被读 | XXXXXXXX |

### 5. 多主控器I2C总线控制（I2C LC）寄存器

多主控器I2C总线控制寄存器及其位分配如表4-73、表4-74所示。

表4-73 多主控器I2C总线控制寄存器

| 寄存器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| I2CLC | 0x7F004010 | 读/写 | I2C总线主控器线控制寄存器 | 0x00 |

表4-74 多主控器I2C总线控制寄存器的位分配

| I2CLC | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| 滤波器有效 | [2] | I2C总线过滤器有效位<br>当SDA端口用于输 入操作，该位 应当置高位。在两倍的PCLK期间，过滤器能预防由于失灵而发生的错误 | 0 |
| SDA 输出延迟 | [1:0] | I2C总线SDA线延迟长度选择位<br>SDA线被以下面的时钟时间延 迟（PCLK）：<br>00:0 时钟 01:5 时钟<br>10:10 时钟 11:15 时钟 | 00 |

## 4.6.4 I2C总线接口应用举例

S3C6410 I2C总线与使用I2C总线的EEPROM芯片KS24C080C连接电路如图4-18所示。KS24C080C作为I2C从设备，其地址为0xA0，S3C6410通过I2C总线对该芯片进行读／写操作。

### 1. I2C总线接口电路

S3C6410的I2C总线与KS24C080C接口电路如图4-18所示。

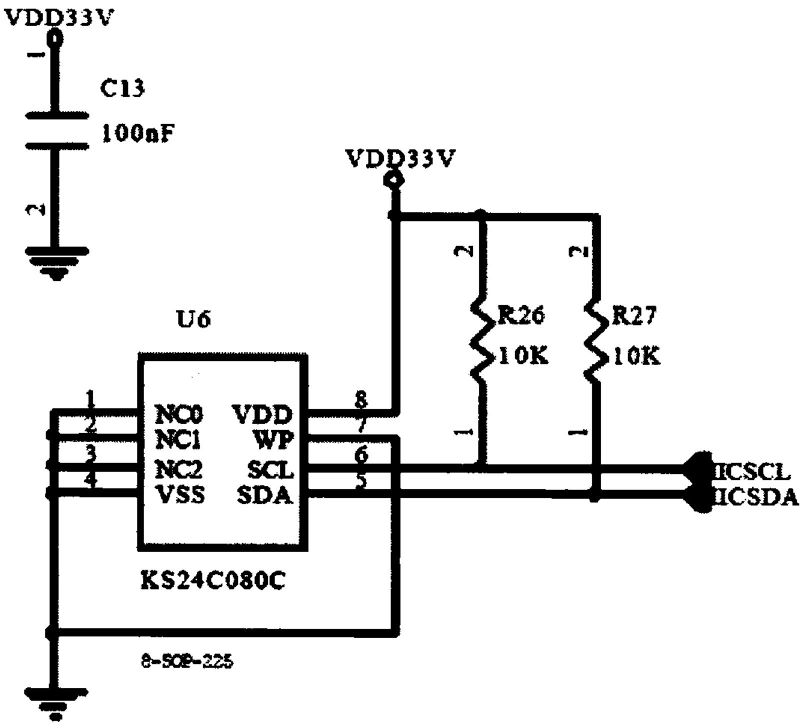



图4-18 S3C6410的I2C总线与KS24C080C接口电路

### 2. I2C总线驱动程序

（1）I2C接口初始化

首先必须进行I2C端口初始化，然后配置I2C控制寄存器。I2C接口初始化操作通过函数void

iic_init()完成。

```
    rGPBCON=(rGPBCON&0xf00fffff)      0x02200000;    //GPB5:IICSCL,GPB6:IICSDA
rGPBPUD= 0x00001400;                                //GPB5、GPB6上拉
    rIICCON=(1<<7) (0<<6)  (1<<5)  (0xf);           //允许中断、fpclk/16、
    TX/RX中断
    rIICADD=0x10;
    rIICSTAT=0x10;                                  //允许TX/RX传输
```

（2）I2C写操作

I2C写操作通过函数void iic_write_24C08（LJ32 slvAddr，U32 addr，U8 data）完成，其中，slvAddr为从设备地址，在本系统中为0xA0，addr为待写入数据到芯片的地址；data为待写入的数据。I2C写操作代码如下所示。

1）填写I2C命令I2C缓冲区数据及大小。

```
iic_command=WRDATA;
iic_data_tx_index=0;
iic_buffer[0]=(U8)daar;
iic_buffer[1]=data;
iic_data_tx_size=2;
```

2）设置从设备地址并启动I2C操作。在I2C轮询函数中进行读／写操作，操作完毕，iic_data_tx_size将被置为-1。

```
rIICDS=slvAddr;
rIICSTAT=0xf0;
```

3）等待写操作完成。

```
while(iic_data_tx_size! = -1);Run_IicPoll();
```

4）等待从设备应答。

```
_iicMode=POLLACK;
    while(1){
    rI2CDS =slvAddr;
    _rI2cStatus =0x100;    //初始化_rIIcStatus,以便检查_rI2cStatus是否改变
    rI2CSTAT =0xf0;        //主发送,产生START信号,有效的接收/发送
    rI2CCON =0xaf;         //继续I2C操作
    while(_iicStatus == 0x100)
        Run_IicPoll();
    if(!(_iicStatus & 0x1))
        break;             //当接收到应答时退出while
    }
    rI2CSTAT =0xd0;        //主发送,产生停止信号,有效的接收/发送
    rI2CCON =0xaf;         //继续I2C操作
    Delay(1);              //等待停止信号有效
```

（3）I2C读操作

I2C读操作通过函数void iic_read_24C08（U32.slvAddr、U32 addr、U8关data）完成，其中slvAddr为从设备地址，在本系统中为0xA0；addr为待读入数据的地址；data为待读入数据的缓冲区指针。I2C读操作代码如下所示。

1）填写I2C命令。

```
iic_command=SETRDADDR;
```

```
iic_data_tx_index=0;
iic_buffer[0]=(U8)daar;
iic_data_tx_size=1;
```

2）等待写操作完成。

```
rIICDS=slvAddr;
rIICSTAT=0xf0;              //主发送,产生START信号,有效的接收/发送
while(iic_data_tx_size! = -1);
```

3）启动I2C操作。在I2C轮询函数中进行读／写操作，操作完毕读取的数据被送入iic_buffer中。

```
_iicMode = RDDATA;
_iicPt = 0;
iic_data_tx_size=1;
rIICDS =slvAddr;
rIICSTAT =0xb0;             //主接收,产生START信号,有效的接收/发送
rIICCON =0xaf;              //继续I2C操作
while(iic_data_tx_size! = -1)
    Run_IicPoll();
*data = _iicData[1];
```

（4）I2C轮询函数

本实例采用轮询方式进行I2C发送和接收处理，包括对POLLACK、RDDATA、WRDATA和SETRDADDR命令的处理。I2C轮询函数的代码如下：

```
void Run_IicPoll(void){
    if(rIICCON & 0x10)   //发送/接收允许中断
void IicPoll(void){
    U32 iicSt,I;
    iicSt = Riicstat;
    if(iicSt & 0x8){}
    if(iicSt & 0x4){}
    if(iicSt & 0x2){}
    if(iicSt & 0x1){}
    switch(_iicMode){
    case POLLACK;
        _iicStatus = iicSt;
        break;
    case RDDATA;
        if((iic_data_tx_size- -) == 0){
            iic_buffer[_iicPt+ + ] = rIICDS;
            rIICSTAT = 0x90;      //主接收,产生停止信号
            rIICCON =0xaf;        //继续I2C操作
            Delay(1);             //等待停止信号有效
            break;
        }
        iic_buffer[_iicPt+ + ] = rI2CDS;
        if((iic_data_tx_size) == 0)
            rIICCON =0x2f;
        else
            rIICCON =0xaf;
        break;
```

```
        case WRDATA;
            if((iic_data_tx_size- -) == 0){
                rIICSTAT = 0xd0;        //主发送,产生停止信号
                rIICCON =0xaf;          //继续I2C操作
                Delay(1);               //等待停止信号有效
                break;
            }
            rIICDS = iic_buffer[_iicPt+ + ]; // iic_buffer[0]已经空
            for(i=0;i<10;i++);
            rIICCON =0xaf;      //继续I2C操作
            break;
        case SETRDADDR;
            if((iic_data_tx_size- -) == 0){
                break;
                }
            rIICDS = iic_buffer[_iicPt+ + ]; // iic_buffer[0]已经空
            for(i=0;i<10;i++);
            rIICCON =0xaf;      //继续I2C操作
            break;
        default;
            }
    }
```

# 4.7 SPI接口

## 4.7.1 SPI总线接口简介

SPI（Serial Peripheral Interface）总线是一种全双工串行同步通信协议，SPI串行外围接口能够支持串行数据传输，SPI包括两个8、16、32位的移位寄存器，分别用于传输和接收。在SPI传输期间，数据同步传输（串行输出）和接收（串行输入）。表4-75列出了SPI和外设之间的外部信号。当不使用SPI总线时，SPI的所有端口可以作为通用I/O端口。

表4-75 外部信号描述

| 名　称 | 方　向 | 描　述 |
|---|---|---|
| XspiCLK | 输入/输出 | XspiCLK是串行时钟，用于控制传输数据的时间 |
| XspiMISO | 输入/输出 | 在主控器模式，该端口是输入端口。输入模式用于从从属器输出端口获得数据。数据通过从属器模式下的端口传输到主控器 |
| XspiMOSI | 输入/输出 | 在主控器模式，该端口是输出端口。该端口用于从主控器输出端口传输数据。数据通过从属器模式下的端口被接收 |
| XspiCS | 输入/输出 | 从属器选择信号，当XspiCS是低电平时，所有数据发送/接收依次被执行 |

## 4.7.2 SPI操作

S3C6410中，SPI在S3C6410和外设之间传输1位串行数据。S3C6410中的SPI支持CPU或DMA分别发送或接收FIFO，并且同时双向传输数据。SPI有两个通道：发送通道和接收通道。

CPU（或DMA）必须在SPI_TX_DATA寄存器写入数据以将数据写入到FIFO。寄存器的数据会自动移动到发送FIFO。为了从接收FIFO读数据，CPU（或DMA）必须访问寄存器SPI_RX_DATA，而数据会被自动发送到寄存器SPI_RX_DATA。

（1）操作模式

HS_SPI有两种模式，主控器模式和从属器模式。在主控器模式下，HS_SPICLK产生，并被发送到外设。PSS选择从属器的信号，当它为低电平时，表示数据有效。在信息包开始发送和传输前PSS必须设置为低电平。

（2）FIFO访问

S3C6410的SPI支持CPU和DMA访问FIFO。CPU访问和DMA访问FIFO的数据大小可以选择8位或者32位。如果选择8位数据大小，有效位为0到7位。通过触发用户定义的阈值，CPU访问正常打开和关闭。每个FIFO的触发器级别被设置从0～64字节。SPI_MODE_CFG寄存器的TxDMAOn或RxDMAOn位必须设置DMA访问。DMA访问仅支持单传输和4个脉冲传输。在发送FIFO时，DMA要求信号为高电平，直到FIFO满。在接收FIFO时，如果FIFO非空，DMA要求信号是高电平。

（3）接收FIFO中的结尾字节

在中断模式下，SPI接收FIFO中采样的数量小于阈值的值，或者是DMA四个脉冲模式，并且没有其他数据被接收，该保留字节被称为结尾字节。为了在接收FIFO中移动这些字节，需要用到内部时钟和中断信号。

基于APB总线时钟，内部时钟的值能被设置到1024个时钟。当定时器值为0时，中断信号发生，并且CPU能在FIFO中移动结尾字节。

（4）信息包数目控制

在主控器模式下，SPI可以控制接收的信息包的数量。如果有信息包要被接收，则需要设置SFR（Packet_Count_reg）有多少包要接收。当包的数量和设置的数量相同时，SPI停止产生SPICLK。在该功能被再次装载前，它严格遵循软件和硬件复位（软件复位能清除除了专用的寄存器外的所有的寄存器；硬件复位清除所有的寄存器）。

（5）nCS控制

nCS可以选择自动控制和手动控制。对于手动控制，Auto_n_Manual必须设置为默认值0，nCS电平设置必须和nSSout位的设置相同。自动控制下，nCS能被固定在包与包之间，Auto_n_Manual设置为1时，只要nCS不活动，nCS_time_count必须设置，这时nSSout是可用的。

### 4.7.3 SPI专用寄存器

专用的寄存器必须以下面的顺序设置（nCS为手动模式）：

1）设置传输类型（CPOL或CPHA设置）。

2）设置时钟设置寄存器。

3）设置SPIMODE设置寄存器。

4）设置APIINT_EN寄存器。

5）如果需要的话，设置信息包数量配置寄存器。

6）设置发送和接收通道启动。

7）设置nSSout为低位来开始发送或接收操作。

设置nSSout为低位，启动发送数据写。如果自动芯片选择位被设置，则不应当控制nCS。专用的寄存器的描述如表4-76～表4-91所示。

表4-76 CH_CFG寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 初始值 |
|---|---|---|---|---|
| CH_CFG（Ch0） | 0x7F00B000 | 读/写 | SPI配置寄存器 | 0x0 |
| CH_CFG（Ch1） | 0x7F00C000 | 读/写 | SPI配置寄存器 | 0x0 |

表4-77 CH_CFG寄存器的位分配

| CH_CFG | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| SW_RST | [5] | 读/写 | 软复位<br>0：不活动1：活动 | 1’b0[1] |
| SLAVE | [4] | 读/写 | SPI通道是主控器还是从属器<br>0：主控器1：从属器 | 1’b0 |
| CPOL | [3] | 读/写 | 确定一个有效的高位或者低位时钟<br>0：有效的高位1：有效的低位 | 1’b0 |
| CPHA | [2] | 读/写 | 从两个基本的不同传输格式中选择一个<br>0：格式A 1：格式B | 1’b0 |
| RxChOn | [1] | 读/写 | SPI接收通道打开<br>0：通道关闭1：通道打开 | 1’b0 |
| TxChOn | [0] | 读/写 | SPI发送通道打开<br>0：通道关闭1：通道打开 | 1’b0 |

表4-78 Clk_CFG寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 初始值 |
|---|---|---|---|---|
| Clk_CFG（Ch0） | 0x7F00B004 | 读/写 | 时钟配置寄存器 | 0x0 |
| Clk_CFG（Ch1） | 0x7F00C004 | 读/写 | 时钟配置寄存器 | 0x0 |

表4-79 Clk_CFG寄存器的位分配

| Clk_CFG | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| ClkSel | [10:9] | 读/写 | 选择时钟源来产生SPI时钟输出<br>00：PCLK 01：USBCLK<br>10：Epll时钟 11：保留<br>对于使用USBCLK源的，USB_SIG_MASK在系统控制器必须设置为打开 | 2’b0 |

[1] 1’b0表示第1位数为0；2’b0表示前2位数为0；3’b0表示前3位数为0；依此类推。——编辑注

（续）

| Clk_CFG | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| ENCLK | [8] | 读/写 | 时钟开/关<br>0：无效 1：有效 | 1'b0 |
| Prescaler Value | [7:0] | 读/写 | SPI时钟输出分频频率<br>SPI时钟输出=时钟源/（2×（预分频值+1）） | 8'h0 |

表4-80 MODE_CFG寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 初始值 |
|---|---|---|---|---|
| MODE_CFG（Ch0） | 0x7F00B008 | 读/写 | SPI FIFO控制寄存器 | 0x0 |
| MODE_CFG（Ch1） | 0x7F00C008 | 读/写 | SPI FIFO控制寄存器 | 0x0 |

表4-81 MODE_CFG寄存器的位分配

| MODE_CFG | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Ch_tran_size | [30:29] | 读/写 | 00：字节 01：半字<br>10：字 11：保留 | 2'b0 |
| Trailing Count | [28:19] | 读/写 | 计数值，从接收FIFO中写入的最后数据到覆盖FIFO中结尾的字节 | 10'b0 |
| BUS transfer size | [18:17] | 读/写 | 00：字节 01：半字 | 2'b0 |
| RxTrigger | [16:11] | 读/写 | 在中断模式，接收FIFO触发电平为<br>6'h0～6'h40。该值是接收FIFO的字节数 | 6'b0 |
| TxTrigger | [10:5] | 读/写 | 在中断模式，发送FIFO触发电平为<br>6'h0～6'h40。该值是发送FIFO的字节数 | 6'b0 |
| reserved | [4:3] | — | — | — |
| RxDMA On | [2] | 读/写 | DMA模式打开/关闭<br>0：DMA模式关闭 1：DMA模式打开 | 1'b0 |
| TxDMA On | [1] | 读/写 | DMA模式打开/关闭<br>0：DMA模式关闭 1：DMA模式打开 | 1'b0 |
| DMA transfer | [0] | 读/写 | DMA传输类型，单个或者四个脉冲<br>0：单个 1：4个脉冲设置DMA传输大小必须和SPI中一样 | 1'b0 |

表4-82 Slave_slection_reg寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 初始值 |
|---|---|---|---|---|
| Slave_slection_reg（Ch0） | 0x7F00B00C | 读/写 | 从属器选择寄存器 | 0x1 |
| Slave_slection_reg（Ch1） | 0x7F00C00C | 读/写 | 从属器选择寄存器 | 0x1 |

表4-83 Slave_slection_reg寄存器的位分配

| Slave_slection_reg | 位 | 读/写 | 描述 | 初始状态 |
|---|---|---|---|---|
| nCS_time_count | [9:4] | 读/写 | nSSout无效时间 = ((nCS时间计数+3)/2) × SPICLKout) | 6'b0 |
| reserved | [3:2] | — | 保留 | |
| Auto_n_Manual | [1] | 读/写 | 芯片选择设置分手动或自动选择<br>0：手动 1：自动 | 1'b0 |
| nSSout | [0] | 读/写 | 从属器选择信号（只用于手动）<br>0：有效 1：无效 | 1'b1 |

表4-84 SPI_INT_EN寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 初始值 |
|---|---|---|---|---|
| SPI_INT_EN（Ch0） | 0x7F00B010 | 读/写 | SPI中断启动寄存器 | 0x0 |
| SPI_INT_EN（Ch1） | 0x7F00C010 | 读/写 | SPI中断启动寄存器 | 0x0 |

表4-85 SPI_INT_EN寄存器的位分配

| SPI_INT_EN | 位 | 读/写 | 描述 | 初始状态 |
|---|---|---|---|---|
| IntEnTrailing | [6] | 读/写 | 用于结尾计数设置为0的中断启动<br>0：无效 1：有效 | 1'b0 |
| IntEnRxOverrun | [5] | 读/写 | 用于接收超限运行的中断启动<br>0：无效 1：有效 | 1'b0 |
| IntEnRxUnderrun | [4] | 读/写 | 用于接收欠载运行的中断启动<br>0：无效 1：有效 | 1'b0 |
| IntEnTxOverrun | [3] | 读/写 | 用于发送超限运行的中断启动<br>0：无效 1：有效 | 1'b0 |
| IntEnTxUnderrun | [2] | 读/写 | 用于发送欠载运行的中断启动。在从属器模式，转向从属器发送路径后，该位必须首先清除<br>0：无效 1：有效 | 1'b0 |
| IntEnRxFifoRdy | [1] | 读/写 | 用于RxFifoRdy（中断模式）的中断启动<br>0：无效 1：有效 | 1'b0 |
| IntEnTxFifoRdy | [0] | 读/写 | 用于TxFifoRdy（中断模式）的中断启动<br>0：无效 1：有效 | 1'b0 |

表4-86 SPI_STATUS寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 初始值 |
|---|---|---|---|---|
| SPI_STATUS（Ch0） | 0x7F00B014 | 读 | SPI状态寄存器 | 0x0 |
| SPI_STATUS（Ch1） | 0x7F00C014 | 读 | SPI状态寄存器 | 0x0 |

表4-87　SPI_STATUS寄存器的位分配

| SPI_STATUS | 位 | 读/写 | 描　　述 | 初始状态 |
|---|---|---|---|---|
| TX_done | [21] | 读 | 表示传输完成<br>0：所有的情况除了熔断情况 | 1'b0 |
| Trailing_byte | [20] | 读 | 表示结尾计数是0 | 1'b0 |
| RxFifoLvl | [19:13] | 读 | 数据水平在接收FIFO<br>0～7'h40字节 | 7'b0 |
| TxFifoLvl | [12:6] | 读 | 数据水平在发送FIFO<br>0～7'h40字节 | 7'b0 |
| RxOverrun | [5] | 读 | 接收FIFO超限错误<br>0：无错误 1：错误 | 1'b0 |
| RxUnderrun | [4] | 读 | 接收FIFO欠载运行错误<br>0：无错误 1：错误 | 1'b0 |
| TxOverrun | [3] | 读 | 发送FIFO超限错误<br>0：无错误 1：错误 | 1'b0 |
| TxUnderrun | [2] | 读 | 发送FIFO欠载运行错误 | 1'b0 |
| RxFifoRdy | [1] | 读 | 0：FIFO中数据少于触发器电平<br>1：FIFO中数据多于触发器电平 | 1'b0 |
| TxFifoRdy | [0] | 读 | 0：FIFO中数据多于触发器电平<br>1：FIFO中数据少于触发器电平 | 1'b0 |

表4-88　SPI_TX_DATA寄存器

| 寄　存　器 | 地　　址 | 读/写 | 描　　述 | 初始值 |
|---|---|---|---|---|
| SPI_TX_DATA（Ch0） | 0x7F00B018 | 写 | SPI发送数据寄存器 | 0x0 |
| SPI_TX_DATA（Ch1） | 0x7F00C018 | 写 | SPI发送数据寄存器 | 0x0 |

表4-89　SPI_TX_DATA寄存器的位分配

| SPI_TX_DATA | 位 | 读/写 | 描　　述 | 初始状态 |
|---|---|---|---|---|
| TX_DATA | [31:0] | 写 | 该区域包含要通过SPI通道发送的数据 | 32'b0 |

表4-90　SPI_RX_DATA寄存器

| 寄　存　器 | 地　　址 | 读/写 | 描　　述 | 初始值 |
|---|---|---|---|---|
| SPI_RX_DATA（Ch0） | 0x7F00B01C | 读 | SPI接收数据寄存器 | 0x0 |
| SPI_RX_DATA（Ch1） | 0x7F00C01C | 读 | SPI接收数据寄存器 | 0x0 |

表4-91 SPI_RX_DATA寄存器的位分配

| SPI_RX_DATA | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| RX_DATA | [31:0] | 读 | 该区域包含通过SPI通道被接收到的数据 | 32'b0 |

### 4.7.4 SPI接口应用举例

本小节介绍SPI在ARM11处理器中的编程实现。结合以上对SPI接口的介绍，读者很容易掌握SPI接口的功能及特性。以下是SPI部分代码的具体实现。

#### 1. SPI复位（功能是复位某一个SPI通道）

```
void SPI_reset(SPI_channel * ch) {
//带有时钟延迟的复位
Outp32( &ch->m_cBase->ch_cfg,Inp32(&ch->m_cBase->ch_cfg) & ~(0x3F) ); //  清除寄存器
Outp32( &ch->m_cBase->ch_cfg,Inp32(&ch->m_cBase->ch_cfg) | (1<<5) );
    Delay(10);
    //释放复位信号
    Outp32( &ch->m_cBase->ch_cfg,Inp32(&ch->m_cBase->ch_cfg) & ~(1<<5) );
```

#### 2. SPI通道初始化（功能是初始化某一个SPI通道）

```
SPI_channel* SPI_channel_Init(int channel) {
    SPI_channel* ch = &SPI_current_channel[channel];
    memset( (void*)ch,0,sizeof(SPI_channel) );
    ch->m_ucChannelNum = channel;
    if(channel == 0) {
        ch->m_cBase = (SPI_SFR*)SPI0_BASE;
        ch->m_ucIntNum = NUM_SPI0;
        ch->m_fDMA= SPI_DMADoneChannel0;
        ch->m_fISR = SPI_interruptChannel0;
#ifdef SPI_NORMAL_DMA
        ch->m_ucDMACon = DMA0;
        SYSC_SelectDMA(eSEL_SPI0_TX,1);         //标准的DMA设置
        SYSC_SelectDMA(eSEL_SPI0_RX,1);         //标准的DMA设置
#else
        ch->m_ucDMACon = SDMA0;
        SYSC_SelectDMA(eSEL_SPI0_TX,0);         //安全的DMA设置
        SYSC_SelectDMA(eSEL_SPI0_RX,0);         //安全的DMA设置
#endif
    }
    else if(channel == 1) {
        ch->m_cBase = (SPI_SFR*)SPI1_BASE;
        ch->m_ucIntNum = NUM_SPI1;
    ch->m_fDMA = SPI_DMADoneChannel1;
        ch->m_fISR = SPI_interruptChannel1;
#ifdef SPI_NORMAL_DMA
        ch->m_ucDMACon = DMA1;
        SYSC_SelectDMA(eSEL_SPI1_TX,1);         //标准的DMA设置
        SYSC_SelectDMA(eSEL_SPI1_RX,1);         //标准的DMA设置
#else
        ch->m_ucDMACon = SDMA1;
        SYSC_SelectDMA(eSEL_SPI1_TX,0);  //标准的DMA设置
```

```
        SYSC_SelectDMA(eSEL_SPI1_RX,0); //  标准的DMA设置
#endif
    }
    else {
        Assert(0);
    }
     Outp32(&ch->m_cBase->slave_sel,Inp32(&ch->m_cBase->slave_sel) | (1<<0) ); //
片选OFF-激活LOW
    SPI_GPIOPortSet(channel);            //  设置通道GPIO
    return ch;
}
```

### 3. SPI基本寄存器的设置

```
void SPI_setBasicRegister(SPI_channel* ch) {
Outp32( &ch->m_cBase->ch_cfg, //清除寄存器
(ch->m_eClockMode<<4)|          //主/从模式
    (ch->m_eCPOL<<3)|           //CPOL高态有效/行
    (ch->m_eCPHA<<2) );         //CPHA传输格式
    Outp32( &ch->m_cBase->clk_cfg, (Inp32(&ch->m_cBase->clk_cfg) & ~(0x7ff))|
                                //清除寄存器
    (ch->m_eClockSource<<9) |                         //时钟设置
( ( (ch->m_eClockMode==SPI_MASTER)?(1):(0) )<<8) |    //时钟使能
    ch->m_cPrescaler);                                //预定标器设置
Outp32( &ch->m_cBase->mode_cfg,(Inp32(&ch->m_cBase->mode_cfg)&(u32)(1<<31)) |
                                //清除寄存器
(ch->m_eChSize<<29)|            //通道传输大小
(ch->m_uTraillingCnt<<19)|      //trailling计数
(ch->m_eBusSize<<17)|           //总线传输大小
(ch->m_ucRxLevel<<11)|          // Rx触发级
(ch->m_ucTxLevel<<5)|           //Tx触发级
(ch->m_eDMAType<<0) );          //DMA类型
}
```

# 4.8 DMA-6410XP实验平台

## 4.8.1 实验平台功能

DMA-6410XP 采用32位ARM11内核的S3C6410处理器，独有的3D，在支持DVFS的同时还拥有低功耗的内存接口特性，对于设计者来说其是一款高效能低价位的处理器。微处理器采用AXI、AHB和APB总线组成的64/34-bit内部总线结构接口，处理能力最大可达667MHz。

DMA-6410XP实验平台不仅提供了完整的底层驱动（开发环境为RVDS 2.2以上），并提供了Windows Embedded CE6.0、Linux 2.6.27及Google Android 1.6下各周边接口的驱动，还提供了三个操作系统下的图形接口驱动范例，可以让你轻松学会嵌入式相关知识。

## 4.8.2 外观及接口

DMA-6410XP实验平台外观如图4-19所示，实验平台界面如图4-20所示。

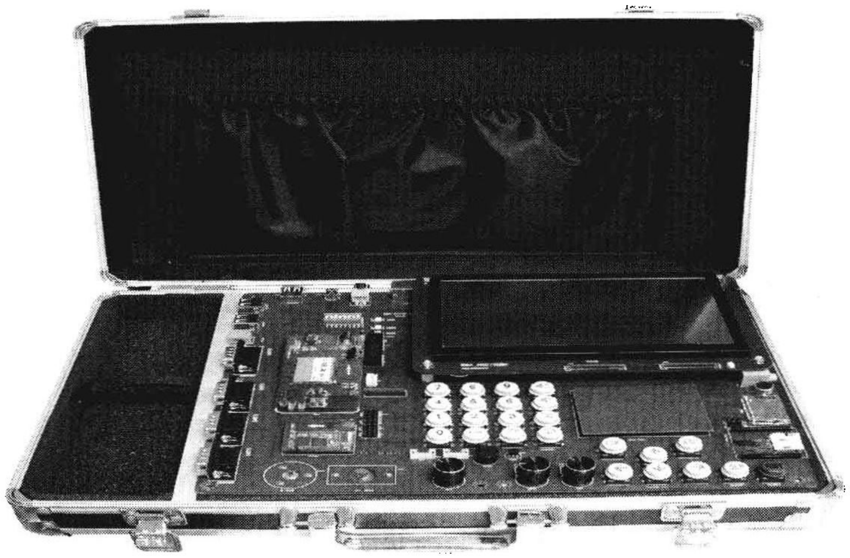


图4-19 DMA-6410XP实验平台

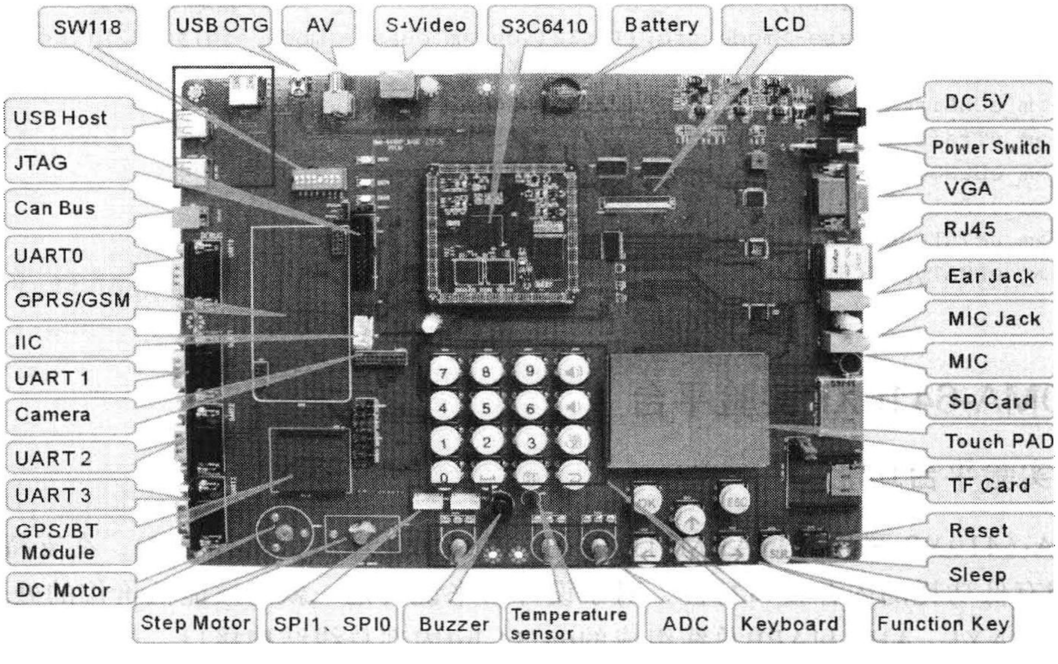



图4-20 DMA-6410XP实验平台界面

## 4.8.3 平台硬件配置

（1）中央处理器

CPU：Samsung S3C6410XHL-66，主频为667MHz，内核为ARM1176JZF-S。

（2）外部内存

- SDRAM内存：平台上提供64MB 2片Mobile DDR SDRAM，共128MB的标准配置。
- NOR Flash：2MB内存。
- NAND Flash：SLC NAND Flash标配为128MB内存；MLC NAND Flash为1GB/2GB/4GB内存（Option）。

（3）网络接口

1个10/100M Ethernet，采用DM9000AE，带指示灯RJ-45界面。

（4）USB界面

3个USB HOST（USB 1.1 规范）接口，支持全速 （12Mbps） 或低速 （1.5Mbps） 传输；1个USB HS OTG（USB 2.0规范）接口，最高支持480Mbps高速传输。

（5）串行口

1个5线制串行口，3个复用串行口（ 与BlueTooth / GPS / GPRS GSM / Touch PAD复用）。

（6）音效接口

采用AC97的接口芯片，立体声音效输出接口可接耳机或音箱；支持录音，底板有音效输入接口可接麦克风。

（7）CMOS Sensor 摄影机接口

底板上内置一个130万像素的CMOS Sensor摄影机，可直接摄影并在液晶屏幕上显示，并有一个2.0mm间距双排插座用于摄影机扩充，这个扩充接口可连接其他型号的CMOS Sensor摄影机，支持的标准为ITU-R BT.601/656 YCBCR 4-bit standard。

（8）LCD界面

- 兼容3.3V / 5V供电LCD屏幕。
- 系统平台标配为16.7M色800×480 / 7.0英寸TFT液晶屏幕，附触控功能。
- 支持TFT液晶屏幕，尺寸从3.5英寸到15英寸，板上可供TFT LCD逆变器电源（+5V）。

（9）Touch界面

1个触控屏幕控制器（四线电阻式）。

（10）VGA界面

可直接连接各种VGA接口的CRT显示器或液晶显示器，带对比度微调电位器，最高输出分辨率为1024×768。

（11）TV（NTSC / PAL）界面

支持TV Tunet（NTSC及PAL），可直接接各种AV接口的CRT显示器。

（12）S-VIDEO界面

1个S-VIDEO视频信号输出。

（13）SD卡界面

1个高速SD / HSMMC和SDIO设备。

- Linux 2.6.27：支援高速SDHC卡（SD 2.0）2GB / 4GB / 8GB / 16GB / 32GB并兼容标准SD 1.1。

- Windows Embedded CE 6.0：支援高速SDHC卡（SD 2.0）2G / 4G / 8G / 16G / 32GByte并兼容标准SD 1.1。

（14）Micro SD（T-Flash）卡界面

1个T-Flash存储设备。

（15）RTC时钟

S3C6410内部整合，外部提供RTC电池，插座安装，可更换。

（16）JTAG界面

1个ARM标准20PIN JTAG接口位于底板上，可支持内核板独立除错，也可通过底板支持RVDS2.2以上版本等除错。

（17）CAN总线接口

1个CAN总线接口，全面支持CAN2.0A和CAN2.0B协议。

（18）Touch PAD

1个笔电触控板，RS-232界面。

（19）马达界面

1个步进电动机，1个直流电动机。

（20）红外通信接口（Option）

采用高灵敏度RPM851A红外收发模块，串行端口接口。

（21）内嵌GPS模块接口（Option）

采用高灵敏度、低耗电量、高性能低价位MTK GPS芯片。具备快速定位及追踪32颗卫星的能力，接上GPS模块后，便成为一台GPS导航仪。

（22）内嵌式蓝牙语音模块接口（Option）

采用了功能齐全的蓝牙语音模块，配合本平台的软件可以实现的功能有：

- 与蓝牙手机电话联机同步；
- 通讯簿下载；
- 简讯下载；
- 来电显示；
- 来电接听；
- 拨打电话；
- 手机拨打免持（通过复用麦克风）。

（23）内嵌式GPRS / GSM通信模块（Option）

采用SIMCom GPRS通信模块，配合本平台的软件可以实现的功能有：

- 拨打电话；
- 收发简讯；
- 透过GPRS上网与服务器通信。

（24）时钟设计

采用无源晶振模式设计。

（25）Reset 电路

采用手动重置和芯片重置相结合的方式，芯片采用MAX811，重置稳定可靠。

（26）电源界面

采用5V、2A外置电源供电，可支持电池工作。

（27）平台其他功能

- 2个除错指示LED。
- 1个电源指示LED。
- 过流保护：采用自恢复保险器件。
- 矩阵键盘：4×4个按键，可进行键盘仿真，5个功能键。
- ADC转换器：8通道12bit精度ADC，其中4路硬件设计为触控荧幕，1路接温度应器（LM35D），3路接可调电阻。
- I2C控制器：2个I2C串行接口，其中1个与CAMERA及EEPROM（AT24C02）接口复用，1个与串行口复用，均为CPU内置，支援400kbps快速及多主模式。
- SPI接口：分别是2通道5线制SPI高速同步串行接口（作为主控设备支持50Mbps的收发，作为从设备支持50Mbps的发、20Mbps的收），其中一个SPI高速同步串行接口与CAN接口控制复用。

## 4.8.4　支持操作系统

支持的操作系统如下：

- Google Android 1.6；
- Linux 2.6.27；
- Windows Embedded CE 6.0。

支持Linux 2.6.27 / Google Android 1.6操作系统及驱动原始程序代码：

- 2MB NOR Flash / 128MB NAND Flash / 128MB DDR SDRAM 驱动。
- 触控LCD驱动（支持分辨率800×480的7英寸及10.2英寸LCD含触控）。
- 10/100MB DM9000AE驱动。
- USB Host / USB Device驱动。
- 普通及高速SD/MMC卡（最大至16GB）和SDIO设备驱动。
- AC97音效录音播放驱动。
- Camera视频驱动方面支持OV9653 CMOS Sensor。
- 4个串行驱动（1个标准串行口，3个BT/ GPS/ GPRS GSM/ TouchPAD复用串行口）。
- I2C、RTC、SPI、ADC、VGA、CAN和RS-485驱动。
- 矩阵键盘、喇叭、麦克风等多种驱动。
- 应用程序包括QT / Android 1.6 等图形系统。
- 3D硬件加速方面提供OpenGLES测试程序驱动Source Code。
- MFC多媒体硬件编译码方面支持H264/H263/MPEG4/WMV9硬件译码，提供驱动及测试

程序驱动Source Code。

- JPEG硬件编译码：支持JPEG图片硬件编译码，提供驱动及测试程序驱动Source Code。
- 上层应用测试程序AP：
- 提供LED明亮控制、键盘输入控制、LCD背光亮度控制、喇叭音量控制、步进电动机、直流电动机、网络通信、蓝牙通信、WiFi无线上网等测试程序AP。
- 提供MP3、MP4、EBook、Photo、录音等多媒体测试程序AP。
- 提供ARM11 6410的特殊多媒体功能等测试程序AP：3D硬件图形加速、JPEG硬件译码、MFC多媒体硬件译码。

# 第5章 Windows CE Boot Loader与OAL

## 5.1 Boot Loader介绍

Boot Loader是在操作系统内核执行之前执行的一段小程序，通过这段小程序，可以初始化硬设备、建立系统的内存空间映像图，从而将系统的软硬件环境带到一个已知的状态，以便为最终调用操作系统内核准备好环境。最终，Boot Loader把操作系统内核映像加载到RAM中，并将系统控制权传递给它。

每种不同的CPU体系结构都有不同的Boot Loader，Boot Loader实际上也依赖于具体的嵌入式板级设备的配置。

Boot Loader是用来管理目标设备启动过程的特定程序，它初始化目标设备硬件，并允许将Windows CE操作系统执行时映像从开发主机下载到Windows CE目标设备的RAM或再将它写入到Flash，并跳转到操作系统的起始点启动操作系统，因此，大多数Boot Loader都包含两种不同的操作模式。

（1）启动加载模式

这种模式是Boot Loader的正常工作模式，Boot Loader从储存介质将操作系统加载到RAM中，并从RAM中启动执行操作系统。

（2）下载模式

Boot Loader从开发主机下载操作系统执行时映像文件先到达目标设备的RAM，然后再写到目标设备的Flash等储存介质中。

为了实现下载模式，开发主机和目标设备之间必须建立一种连接，包括以太网连接、USB连接或串行端口连接等，Boot Loader使用这种连接从开发主机下载操作系统执行时映像到目标设备。为了实现启动加载模式，目标设备必须有一个本地的永久储存器，可以是CF卡、硬盘、Flash内存或EEPROM等。

一旦操作系统被加载，Boot Loader就可以用来监视或调试目标设备。由于Boot Loader仅用在开发过程中，所以在定制操作系统时没有必要将它包含在操作系统执行时映像中。

Boot Loader除了基本功能之外，OEM或者开发者可以添加更多的功能到Boot Loader，如将控制权直接传递给位于Flash的Windows CE操作系统或内存读/写测试等。

### 5.1.1 Boot Loader的引导选项

Boot Loader的基本功能是初始化设备硬件、控制引导过程和下载操作系统执行时映像到目

标设备的RAM或Flash内存。除了这些基本功能之外，还需要一些辅助功能，如设置目标设备IP地址、超时时间等。另外，开发者可能根据实际需要设计一些附加的、灵活的功能。基本功能、辅助功能和附加功能融为一体，构成了一个Boot Loader的功能选项。为了能对Boot Loader的功能选项进行选择和控制，需要建立开发主机和目标设备之间的Boot Loader通信。

（1）Boot Loader通信

开发主机与Windows CE目标设备通过以太网和串行端口连接在一起。开发主机上的超级终端通过串行端口连接向目标设备发送命令，实现对Boot Loader的控制，同时目标设备上的Boot Loader通过串行端口连接发送调试和执行信息到超级终端窗口。以太网连接主要用于下载操作系统执行时映像到目标设备，和对目标设备上正在执行的操作系统映像进行调试。

（2）Boot Loader功能

一个典型的Boot Loader具有多个功能，可以以多种方式加载和引导一个操作系统执行时映像。在设计实现一个Boot Loader时，目前比较常用的方法是设计一个序列化的菜单。当打开目标设备电源或重新开机时，Boot Loader首先开始执行，并通过串行端口在开发主机的超级终端窗口显示这个序列化的菜单，当开发者选择一个功能选项时，Boot Loader就会从开发主机接收到选择这个选项的命令字符，处理它并完成必要的Boot Loader动作。

如何设计Boot Loader的菜单，取决于需要Boot Loader都具有哪些功能，典型的功能包括：利用以太网接口、并行端口或USB接口下载操作系统映像文件、从Flash内存引导操作系统映像、RAM测试、I/O接口测试等。图5-1所示是一个典型的Boot Loader菜单。开发者可以修改这个菜单，添加和删除功能，以满足特定的需要。一般情况下，Boot Loader需要一个超时设置选项，当开发者在设定时间内没有选择任何选项时，Boot Loader执行默认选项功能。

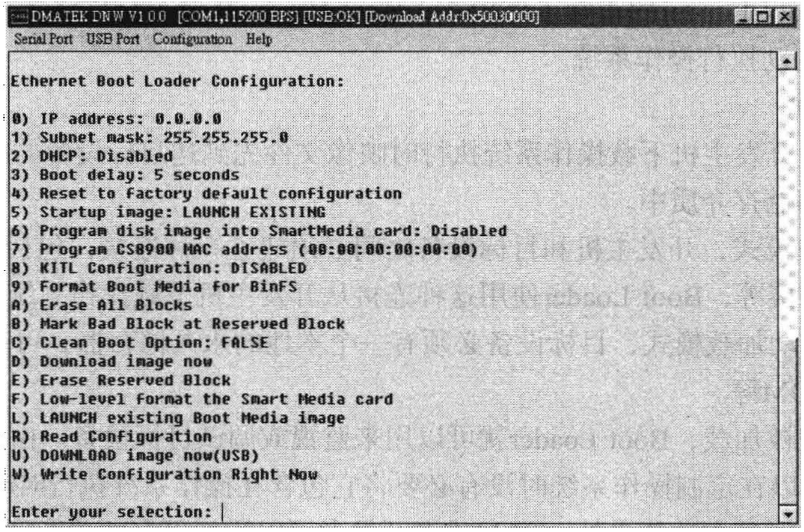



图5-1 Boot Loader的菜单

## 5.1.2 Boot Loader的典型特征

通常Boot Loader驻留在目标设备的永久储存器上，当打开系统电源或重新启动时，Boot

Loader自动执行。当第一次将Boot Loader放置到目标设备的永久储存器上时，开发者需要借助如JTAG这样的模拟工具将Boot Loader写入到Flash等内存。一旦第一次写入成功，Boot Loader应该提供更新自身及下载和写入新的操作系统执行时映像的功能。

在为一个新的硬件平台设计实现一个Boot Loader时，建议开发者考虑实现下列典型功能，对于硬件不支持的功能特征项可以除外。

- Boot Loader应该驻留在Flash等永久储存器里，且最好能够在线更新，另外，应该锁定Boot Loader所在的Flash块，防止Boot Loader被意外损坏。
- Boot Loader应该提供一种消息反馈机制，为开发者或用户提供Boot Loader当前状态、错误及进度等信息。目前典型的方法是通过串行端口连接，向开发主机的超级终端输出这些信息。
- 开发者应该尽量使用微软提供的支持库，以减少开发工作量，并便于后续更新。
- Boot Loader应该不仅能够加载Boot Loader映像和操作系统执行时映像，而且要能从RAM加载，也能从Flash等永久储存器载入。
- Boot Loader应该对下载的数据进行校验，在保证下载的数据准确无误后，再将它写入Flash内存。
- Boot Loader和OAL应该共享相同的硬件初始化程序代码，因为它们完成相同的硬件初始化工作。

除了上述典型功能特征之外，也建议开发者考虑下列附加的功能特征：

- 在Boot Loader的输出信息中，明确显示Boot Loader的版本、开发者及建构日期等信息。
- 设计一个序列化的功能选项菜单，允许开发者或用户对使用动态IP还是静态IP等进行选择和设置，并能够将这些信息进行永久保存，避免每次Boot Loader执行时，都要设置它们。
- 如果硬件允许，Boot Loader应该提供一种从Platform Builder下载自身，并将自身写入Flash内存，以实现更新自身的机制，尽量避免使用其他工具、特殊连线或者协议等。
- Boot Loader应该与Platform Builder的一些开关协调工作，实现重新开机下载、复位跳转，以及发现Boot Loader映像变化时，自动下载Boot Loader等。
- 提供一种监视、反馈机制，当发现下载失败或Boot Loader映像被损坏时，再次下载或更新Boot Loader。另外也可以实现应用JTAG的Flash烧录。
- 如果Boot Loader的文件大小允许，尽可能在Boot Loader映像文件中包含用户可能用到的以太网调试驱动。

在进行具体的Boot Loader设计时，开发者可以根据自己的硬件平台和具体需要，选择实现上述的功能特征，并添加其他必要的功能特征。

### 5.1.3 Windows CE中Boot Loader为可选项

Boot Loader并不是Windows CE嵌入式软件系统的必不可少的组成部分，但是在系统中有Boot Loader可以给嵌入式产品的开发调试工作带来许多便利。在两种情况下嵌入式的软件系统中可以省去Boot Loader。一是目标产品的功能简单，以至于操作系统映像足够小，或者嵌入式硬件设计可以不计成本，大容量存储使用ROM储存设备——目前最广泛使用的是Nor型Flash内存来

做储存，此时可将整个Windows CE的操作系统映像文件全部置于Nor Flash储存中，系统开机，即从Nor Flash中的Windows CE OAL起始位置开始执行。操作系统完全使用XIP（execute In Place，本地执行），或者在Windows CE的OAL中设置功能程序代码，将操作系统映像由Nor Flash复制到RAM。前者因Nor Flash的读取速度远远不如RAM而影响系统的执行速度；后者的执行速度虽可改善，但是增加了将操作系统由Flash复制到RAM的时间，对系统的启动速度影响很大。

第二种不使用Boot Loader的情况是嵌入式目标系统中使用目前已广泛使用的“混合”模式的Flash储存技术，以改善Nand型Flash储存设备不能线性寻址的问题。以S3C6410芯片为例，芯片内置了一个Nand Flash控制器，其中实现了一种被三星公司命名为“阶石”（Stepstone）的技术。就是在系统开机时，把Nand Flash内存的起始8KB的储存内容映像到Nand Flash控制器内部的一个CPU可直接线性寻址访问，并且可以设为系统开机或重新启动时，自动开始执行的8KB大小的缓冲区内。这样Windows CE的软件开发者可以在操作系统映像的前4KB范围内设置Windows CE的映像由Nand Flasht复制到RAM的程序代码，要改变映像文件是只需以烧写的方式或通过JTAG仿真器将它写入Nand Flash储存中。系统上电以后在最初8KB空间范围内的程序代码执行过程中将操作系统映像由Nand Flash复制到RAM并且跳转RAM中执行。

Boot Loader存在于软件系统中，可以给嵌入式产品的开发和调试带来很多方便。最直接的一个便利是每次修改了操作系统映像以后，可以通过以太网或串行端口，或者其他任何一种Boot Loader支持的嵌入式硬件端口将映像文件下载到目标设备。如果没有Boot Loader，那就只能每次烧写Flash，在操作系统更改频繁的情况下，这会增加开发时间，延缓开发工作的进度。

## 5.2 Windows Boot Loader的软件框架和支持库

### 1. Boot Loader的架构

一个典型的以太网Boot Loader是由Blcommon、OEM程序代码、Eboot和网络驱动程序等组成的，Blcommon是通用的BootLoader框架；OEM程序代码是与目标硬件相关的初始化和扩展程序；Eboot是以太网功能程序，如UDP、DHCP、TFTP程序等；EDBG驱动程序是调试以太网卡的驱动程序；Bootpart是用于永久储存器的分区管理程序；FMD是用于Samsung/Sandisk的NAND Flash或Intel StrataFlash（NOR Flash）的Flash管理程序，如图5-2所示。

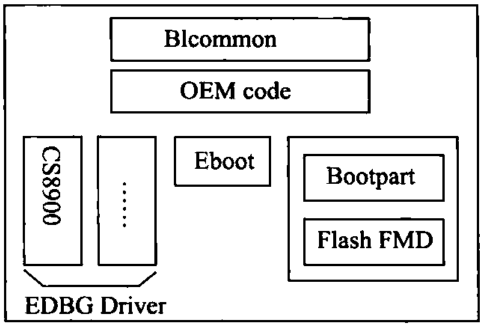


图5-2 Boot Loader的结构

Blcommon、Eboot和网络驱动程序是可以被重复使用和可移植的程序代码库，微软提供了它们的原始程序代码，这部分程序代码与OEM程序代码一起进行编译后，构成最终的Boot Loader二进制映像。开发者在进行Boot Loader的开发时，一般只需要编写OEM程序代码。

在Windows CE中，Blcommon程序代码库一般放置在%_WINCEROOT%Public\Common\Oak\Drivers\Ethdbg\Blcommon文件夹中，用于实现Boot Loader的基本框架，其功能主要包括：重新定位Boot Loader在RAM中的位置、译码.bin文件的内容、验证校验及监视加载过程等。

Blcommon程序代码库的入口点为BootLoaderMain函数，当开启目标系统电源或重新启动时，它由Boot Loader的Startup程序代码调用。

OEM程序代码是与设备初始化及硬件扩展有关的程序代码，是OEM或开发者根据自己的目标硬件需要编写的程序代码，或者是在微软提供的范例程序代码的基础上需要根据自己的目标硬件更改的程序代码。其中设备初始化程序代码由Boot Loader与OAL共享，一般将OEM程序代码放置到目标硬件的BSP中。在Windows Embedded CE 6.0中，一般将Boot Loader的OEM程序代码放置到%_WINCEROOT%\Platform\<Hardware Platform Name>\Src\ Bootloader\Eboot文件夹中（其中Hardware Platform Name是根据读者使用的嵌入式装置，寻找相对应的目录，例如：DMA6410XP，之后描述的文件夹都是用这种方式对应，此文件夹在随机附赠的光盘中可以找到）。

Eboot程序代码库一般被放置在%_WINCEROOT%public\Common\Oak\ Drivers\Ethdbg\ Eboot文件夹中，它为Boot Loader提供DHCP、TFTP和UDP服务，其中DHCP用于动态配置目标硬件以太网卡的IP地址，TFTP用于从开发主机的Platform Builder下载Windows CE操作系统执行时映像，UDP用于开发主机与目标硬件之间的以太网通信。

Bootpart程序代码库一般位于%_WINCEROOT%public\Common\Oak\ Drivers\Ethdbg\ Bootpart文件夹，它辅助Boot Loader进行分区储存、储存设备读/写等操作和管理。Bootpart程序代码与底层的Flash内存存取程序代码一起工作，用于在同一个Flash内存上新建具有一个或多个BinFS（二进制ROM映像文件系统）分区和其他文件系统的分区。其他文件系统分区可以是FAT分区、TFAT分区，或者Boot分区。其中BinFS和Boot分区是只读分区，BinFS分区用于储存.bin文件，Boot分区用于储存引导参数。

EDBG驱动程序库位于%WINCEROOT%public\Common\Oak\Drivers\ Ethdbg文件夹，它封装了许多常见的以太网卡的驱动程序程序代码，并为Boot Loader和操作系统存取提供了通用接口。Boot Loader利用这些接口下载Windows CE操作系统执行时映像，操作系统则利用这些接口实现与Platform Builder的KITL连接。Windows Embedded CE 6.0提供了3C90x、AMDAM79C97x、CS8900A、NS DP83815、NE20000、Realtek RTL8139等调试以太网卡的驱动程序。

### 2. BootLoader的组件

一般情况下，Boot Loader是由OEM Startup Code和Main Code两部分程序代码组成的。

OEM Startup Code是首先执行的Boot Loader程序代码，它初始化内存缓存器、设置CPU频率，并初始化缓存（Cache），它也可以完成一些附加的初始化功能，然后跳转到由%_WINCEROOT%\Platform\<Hardware Platform Name>\ Src\BootLoader\Eboot\ Main.C实现的Main入口点。OEM Startup Code一般由汇编程序来实现。

Main Code是管理Boot Loader执行的主要C语言程序代码，它通过接收来自串行端口的用户命令或者检查相应的硬件开关来决定完成什么操作，也可以在Main Code中实现一个OEM启动画面，当Boot Loader执行时，显示OEM的公司标志（Logo）、Boot Loader版本或固件（firmware）等信息。

Main Code主要包括表5-1所示的程序代码区域。

表5-1 Main Code的主要程序代码区域

| 程序代码区域 | 描 述 |
|---|---|
| 映像下载程序代码 | 应用以太网连接或并行端口连接下载操作系统执行时映像的程序代码 |
| 以太网I/O程序代码 | 通过目标设备与开发主机的以太网连接读/写数据的程序代码，具体实现细节可参考%_WINCEROOT%\public\common\oak\DRIVERS\ETHDBG\EBOOT文件夹 |
| 调试串行端口I/O程序代码 | 一组允许对串行端口进行读、写的函数，主要用于接收用户命令并发送调试输出信息到开发主机的超级终端窗口或Platform Builder的Output窗口。为了简化调试串行端口I/O程序代码的实现，目标设备串行端口应该使用标准的UART，标准的UART应该是16550兼容 |
| Flash写入程序代码（可选） | 将数据写入到Flash内存的一组函数，只有当从Flash内存加载Windows CE操作系统映像时才需要实现这些函数 |
| Firmware监视器（可选） | 提供调试支持的子程序，如内存转储、反编译程序代码等。一旦Boot Loader工作正常，就可以通过发送Firmware监视命令到调试串行端口的监视端口来完成基本的调试 |

### 3. BootLoader的控制流程

Boot Loader控制流程如图5-3所示。

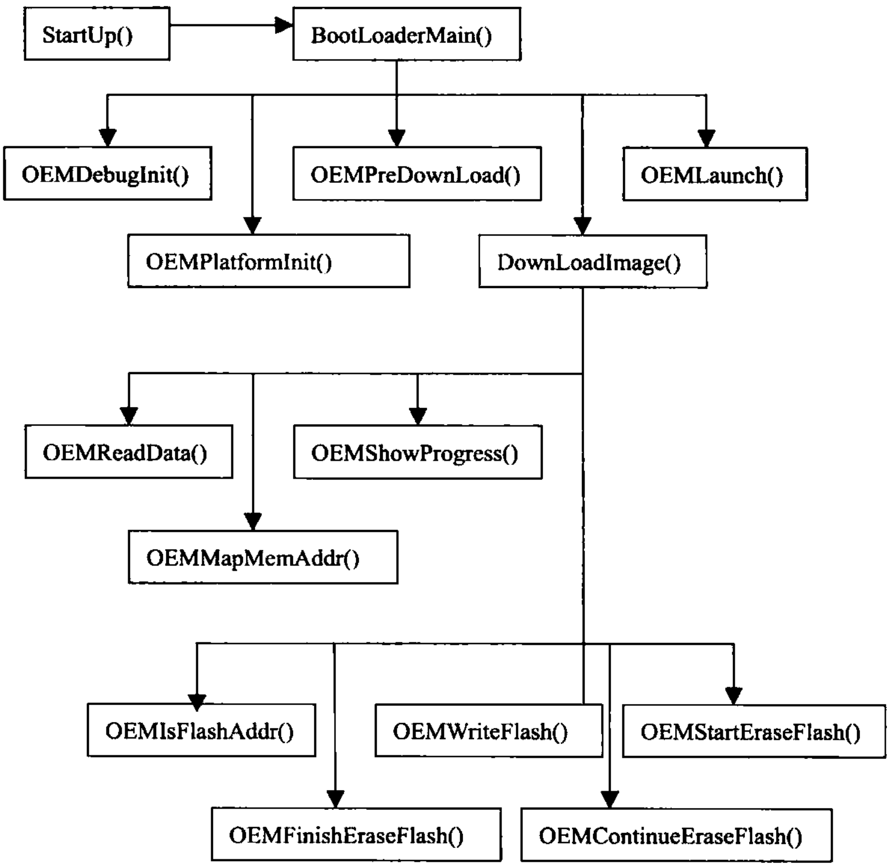



图5-3 Boot Loader的控制流程

当打开电源或启动时，CPU首先执行StartUp函数，StartUp函数通常由汇编程序码编写，主要来建立内存读取和初始化缓存。然后StartUp函数跳转到Blcommon框架的BootLoaderMain函

数。程序代码位于%_WINCEROOT %\Public\Common\Oak\Drivers\ Ethdbg\ Blcommon文件夹，它基本上为所有使用的Boot Loader提供一个标准的控制流程。简化过的程序代码如下所示：

```
void BootloaderMain(void)
{
     //relocate globals to RAM; 把整体变量定位到RAM
    if(!KernelRelocate(pTOC))
    {
        //spin forever; 异常处理
        HALT(BLERR_KERNELRELOCATE);
    }

    //1)初始化调试端口,然后就可以使用OEMWriteDebugString函数了
    if(!OEMDebugInit ())
    {
        //spin forever
        HALT(BLERR_DBGINIT);
    }

   //2)初始化平台 (时钟、驱动和传输)
    if(!OEMPlatformInit ())
    {
        HALT(BLERR_PLATINIT);
    }
     //3)调用OEM编写的OEMPreDownload决定是否下载
    switch(dwAction = OEMPreDownload ())
    {
    case BL_DOWNLOAD:
        //4)下载映像
        if(!DownloadImage(&dwImageStart, &dwImageLength, &dwLaunchAddr))
        {
            //已报错
            SPIN_FOREVER;
        }
        bDownloaded = TRUE;

        //检查所下载映像的签名
           if(*(LPDWORD)OEMMapMemAddr(dwImageStart,dwImageStart + ROM_SIGNATURE_
OFFSET) == ROM_SIGNATURE)
        {
                dwpToc = *(LPDWORD)OEMMapMemAddr(dwImageStart,dwImageStart + ROM_
SIGNATURE_OFFSET + sizeof(ULONG));
            //再一次映像内容
            dwpToc = (DWORD)OEMMapMemAddr(dwImageStart,dwpToc + g_dwROMOffset);
         }
        //没有break,下载后直接继续执行
    case BL_JUMP:
         //在启动操作系统映像前,再次检查签名,如果没下载,那么OEMLaunch()函数应该有能力启动存放
         //在本地储存中的映像
     if(g_bBINDownload && g_pOEMCheckSignature)
        {
            if(!g_pOEMCheckSignature(dwImageStart,g_dwROMOffset,dwLaunchAddr,bDownloaded))
                HALT(BLERR_CAT_SIGNATURE);
        }
        //5)最终启动操作系统映像,此函数不会返回
        OEMLaunch(dwImageStart,dwImageLength,dwLaunchAddr, (const ROMHDR *)dwpToc);
        //不应该执行到这里
```

```
            default:
            //异常
            HALT(BLERR_INVALIDCMD);
        }
    }
```

BootLoaderMain函数中调用的主要函数包括OEMDebugInit、OEMPlatformInit、OEMPreDownLoad、DownloadImage和OEMLaunch，它们一般都被定义在%_WINCE ROOT%Platform\-<Hardware Platform Name>\Src\ BootLoaderkEboot\文件夹的main.c文件中。

下载函数由DownloadImage函数调用，下载函数包括OEMReadData、OEMShowProgress和OEMMapMemAddr。Flash函数也是由DownloadImage函数调用的，当下载的映像需要被写入Flash内存时，DownloadImage调用Flash函数。Flash函数包括OEMIsFlashAddr、OEMWriteFlash、OEMStartEraseFLash、OEMFinishEraseFlash和OEMContinueEraseFlash。下载函数和Flash函数一般在%_WINCEROOT%\Platform\<Hardware Platform Name>\Src\BootLoaderkEboot文件夹下的main.c和flash.c文件中定义。

表5-2为对图5-3中的所有函数的总结性描述，函数都是需要开发者编写的Boot Loader函数。

表5-2 Boot Loader的主要函数

| 函 数 | 描 述 |
|---|---|
| OEMDebugInit | 初始化调试端口（通常为调试串行端口） |
| OEMPlatformInit | 完成平台特定的初始化，包括实时时钟、Flash内存、网络适配卡等 |
| OEMPreDownLoad | 为平台创建一个平台名，获得动态IP地址或分配一个静态IP地址，初始化TFTP传输 |
| DownLoadImage | 下载操作系统映像到目标设备的RAM或Flash内存 |
| OEMLaunch | 加载操作系统执行时映像 |
| OEMReadData | 用于在下载过程中从传输端口读取数据，如果是以太网下载，它会接着调用EbootEtherReadData函数，EbootEtherReadData函数再去调用OEMEthGetFrame函数 |
| OEMShowProgress | 利用LED或其他方式，为用户显示下载进度 |
| OEMMapMemAddr | 用于将Flash内存地址映像为一个在RAM中的位置，用于在RAM中临时储存要写入到Flash内存中的操作系统执行时映像 |
| OEMIsFlashAddr | 用于决定将正在下载的.bin文件储存到Flash内存，或是RAM，如果它所传递的地址是一个Flash地址，那么返回为TRUE，.bin文件将被写入Flash内存 |
| OEMWriteFlash | 将临时缓存在RAM里的操作系统映像写入到Flash内存的最终位置，并验证写操作确实成功完成 |
| OEMStartEraseFlash | 用于初始化Flash内存的擦除过程，使Boot Loader知道储存操作系统映像需要多大的Flash内存空间 |
| OEMFinishEraseFlash | 在操作系统执行时映像下载完毕后，结束Flash内存的擦除过程，保证Flash内存被完全擦除，并准备开始将映像文件写入到Flash内存 |
| OEMContinueEraseFlash | 在.bin文件的每个记录下载过程中，周期性地被调用，以便继续Flash内存的擦除操作，直到Flash内存被完全擦除 |

## 5.3 Windows CE操作系统映像文件格式

被BootloaderMain调用的DownloadImage函数负责从开发主机计算机处下载操作系统映像

到嵌入式目标设备，它是一个BLCOMMON库函数，不是OEM函数，实现原始程序代码在 %_WINCEROOT%\PLATFORM\COMMON\ SRC\COMMON\ BOOT\BLCOMMON\blcommon.c源文件中。尽管这个函数的原始程序代码由微软提供，但是用户需要实现几个OEM函数供它调用，从以太网端口（或者其他外设端口）读取操作系统映像数据。下面介绍这个不需OEM用户开发实现的DownloadImage函数的原始程序代码，目的是从中获取Windows CE操作系统，也包括Boot Loader自身所使用的映像文件的格式信息，这些信息对于用户的开发和调试工作会有所帮助。

由Windows CE的开发工具Platform Builder编译生成的Windows CE操作系统及Boot Loader的映像文件主要有两种格式类型。以.bin为文件名后缀的记录型映像文件和以.nb0为后缀的原始型（raw）映像文件；前者以记录（Record）为单位组织映像的数据，后者则是映像在嵌入式系统中执行时的二进制数据。

DownloadImage函数执行的第一步是调用GetImageType函数获知映像文件的格式类型。

GetImageType也是BLCOMMON库函数，与DownloadImage函数在同一源文件中实现。GetImageType函数查询映像文件格式类型的原理十分简单：每一个Windows CE映像文件在文件数据的起始位置都有一个7字节的特征码（magic number），它与映像文件的格式一一对应如下：

```
"N000FF\x0A" ----BL_IMAGE_TYPE_MANIFEST
"X000FF\x0A" ---- BL_IMAGE_TYPE_MULTIXIP
"B000FF\x0A" ----- BL_IMAGE_TYPE_BIN
"S000FF\x0A" ------ BL_IMAGE_TYPE_SIGNED_BIN
"R000FF\x0A" ------ BL_IMAGE_TYPE_SIGNED_NB0
无特征码----BL_IMAGE_TYPE_UNKNOWN
```

以上的前4种类型都是记录型映像格式，只有BLIMAGE_TYPE_SIGNED_ NB0代表原始型映像文件格式，此外，DownloadImage函数把无特征码的BL_IMAGE_TYPE_UNKNOWN类型也当做原始型映像来处理。BL_IMAGE_ TYPE_MULTIXIP类型在Windows CE 5.0及其以前的版本中用于表示多区段的映像文件，在Windows Embedded CE 6.0版本中使用BL_ IMAGE_TYPE_MANIFEST类型处理多区段的操作系统映像，不再支持BL_IMAGE_TYPE_MULTIXIP类型，仅为了在原始程序代码中保持向下兼容而保留该类型的定义。

BL_IMAGE_TYPE_MANIFEST类型的映像文件的数据是多区段的记录型映像文件的区段（Region）信息。多区段的映像，就是操作系统或者Boot Loader的执行时二进制数据分散在不连续的物理储存区间。Manifest的字面含义是“载货单”，该类型的映像文件并不是真正的Windows CE操作系统的或者Boot Loader的二进制执行时的数据，只是供下载多区段型映像所使用的头文件信息。由于映像数据的性质特殊，BL_IMAGE _TYPE_MANIFEST类型的映像文件也称为Manifest数据，与一般的映像文件数据的_Manifest存放位置不同，它使用专门的DownloadManifest型整体变量g_DownloadManifest来存放。

DownloadManifest结构体类型在头文件%_WINCEROOT%\PUBLIC\ COMMON\ OAK\INC\blcommon.h中的定义如下：

```
typedef struct _MultiBINInfo_
{
   DWORD    dwNumRegions;  // 要下载的文件数
   RegionInfo Region[BL_MAX_BIN_REGIONS];
} MultiBINInfo, *PMultiBINInfo;

typedef MultiBINInfo DownloadManifest;
```

多区段意味着多文件，即一个操作系统或者Boot Loader的二进制执行时数据存放在多个映像文件中，这多个映像文件的数据对应于可以连续，也可以不连续的多个物理储存区域。

DownloadManifest结构体的成员dwNumRegions就是多区段映像的区段个数，也即是后续多个映像文件的文件个数。Region数组成员负责具体存放各区段的信息，BL_MAX_BIN_REGIONS在头文件%_WINCEROOT%\PUBLIC\ COMMOM\OAK\INC\blcommon.h中定义为25，意味着多区段的映像区段数最大不得超过25。Region数组成员的元素类型RegionInfo又是一个在头文件blcommon.h中定义的结构体：

```
typedef struct _Regionlnfo_
{
       DWORD dwRegionStart;
       DWORD dwRegionLength;
       CHAR  szFileName[MAX_PATH];
} RegionInfo,  *PRegionlnfo;
```

Regionlnfo结构体的3个成员dwRegionStart、dwRegionLength、szFileName的含义分别是一个映像区段在物理储存中的起始地址、以字节为单位的区段长度和与映像区段相对应的映像文件文件名。

对于BL_IMAGE TYPE_MANIFEST类型的映像文件，文件数据中紧随7字节的格式类型特征码，之后从下载端口读出的是4字节的校验数据，后面是4字节的区段个数。区段个数数据理所当然存放在g_DownloadManifest整体变量的dwNumRegions成员中，随后依序读出的是映像的g_DownloadManifest. dwNumRegions各区段的属性数据——起始地址、区段长度、区段数据的映像文件名，按顺序存放在g_DownloadManifest.Region数组的各个元素中。DownloadImage函数调用CheckImageManifest函数读取4字节校验数据和随后的全部Manifest数据，然后CheckImageManifest函数调用VerifyChecksum函数对Manifest数据执行校验，不过校验范围仅限于存放在g_DownloadManifest. Region数组中的区段头信息的数据，不包括4字节g_DownloadManifest. dwNumRegions的数据。

BL_IMAGE_TYPE_MANIFEST类型在所有的映像文件格式类型中比较另类，它只是供下载其他类型的映像文件而使用的前导信息，本身并不包含有效的Windows CE操作系统或BootLoader的二进制执行时数据。如果操作系统的映像只有一个区段，或者是使用自身包含有储存位置等头文件信息的记录型映像文件格式，也可以不使用Manifest前导资料。

BL_IMAGE_TYPE_MANIFEST类型的映像文件下载完成以后，Download- Image函数并不立即返回，还要依据整体变量g_DownloadManifest中的Manifest数据所提供的信息继续下载装载有真实的操作系统二进制数据的映像文件。

BL_IMAGE_TYPE_BIN类型是最普通、最常用的记录型映像文件格式，该类型的映像文件均以.bin为文件名后缀，DownloadImage函数调用DownloadBin函数处理这种类型的映像文

件，DownloadImage也是一个BLCOMMON库函数。在BL_IMAGE_TYPE_BIN类型的映像文件中，紧跟着最初的7字节特征码，之后是该映像文件在嵌入式系统中的目的地的物理储存起始地址和以字节为单位的长度，各占4字节，这两个数据的含义分别对应于RegionInfo结构体的dwRegionStart、dwRegionLength两成员。接下来的便是具体存放操作系统二进制数据的、拥有相同结构的多条记录（Record），每条记录由4字节的内存起始地址dwRecAddr、4字节的记录长度dwRecLen、4字节的校验码dwRecChk和dwRecLen各字节的记录数据组成。DownloadImage函数依次从下载端口读出各条记录，并且对它们执行校验，校验也是以记录为单位，待校验无误以后，将去除头信息的记录数据存放到dwRecAddr指定地址的物理储存位置。

如果记录的目的储存位置是Flash储存器，则通常要先缓存在RAM内存中，待整个映像文件全部下载完，再一起写入Flash。

与BL_IMAGE_TYPE_BIN类型相似的最普通、最常用的原始型（Raw）映像文件以.nb0为文件名后缀，它们使用DownloadNB0函数进行处理。由于原始型映像文件的数据只是物理储存中的Windows CE操作系统或者Boot Loader的二进制执行时数据的一个最直接的缓存，所以它是完全无格式的，甚至都没有7字节的特征码，在GetImageType函数中将这种映像文件格式类型定义为BL_IMAGE TYPE_UNKNOWN。通常情况下，DownloadImage函数将无特征码的BL_IMAGE_TYPE_UNKNOWN类型都当做原始型映像文件处理，但是在安全性要求较高的情况下（定义了宏SECURE_BOOTLOADER）无特征码的映像文件是不应该被Boot Loader接受的。原始型映像文件的处理方法是最简单的，Boot Loader只需从下载端口中读出全部的映像数据并且存放到相应的储存位置，而且无须处理校验。原始型映像文件的数据在物理储存中的起始位置地址和长度都取决于Manifest数据，因而.nb0型的映像文件不能单独下载，Boot Loader必须先下载作为它的前导属性信息的BL_IMAGE_TYPE_MANIFEST类型的映像文件。

BL_IMAGE_TYPE_SIGNED_BIN和BL_IMAGE_TYPE_SIGNED_NB0是两个带有数字签名的Windows CE映像文件格式类型，用在对Boot Loader的安全性要求比较高的场合。DownloadImage函数分别调用函数DownloadSignedBin和DownloadSignedNB0对它们进行处理，被调用的这两个函数也都是BLCOMMON库函数。

BL_IMAGE_TYPE_SIGNED_BIN类型的映像文件是基于BL_IMAGE_ TYPE_SIGNED_BIN类型的，在它的7字节的校验码“S000FFx0A”之后是4字节的物理储存起始地址和4字节的映像长度，这与BL_IMAGE_TYPE_ BIN类型是完全相同的。BL_IMAGE_TYPE_SIGNED_BIN类型也使用记录来组织映像的数据，与BL_ IMAGE_TYPE_BIN类型不同的是，在它的记录之上还有一级在原始程序代码中称为结构体（PACKET）的数据封装，结构体是在头文件%_WINCEROOT%\ PUBLIC\COMMON\OAK\INC\sbl.h中定义的一个结构体：

```
typedef struct _PACKETDATA {
    const BYTE * pbData;          //校验数据
    DWORD dwDataLength;           //数据字节长度
    const BYTE * pbSig;           //签名验证
    DWORD dwSigLength;            //签名字节长
    DWORD dwRecAddress;           //记录地址
    DWORD dwRecLength;            //记录长度
```

```
    DWORD dwRecCheck;            //记录检验
    BYTE  bRandomSeed[RANDOM_SEED_LENGTH];  //全部映像文件随机种子
    DWORD dwSequenceNumber;      //结构体的序号
    BYTE  bFlags;                //记录结构体的属性
} PACKETDATA;
```

PACKETDATA结构体的抽象意义是使用数字签名对映像的数据进行校验的单元。两个成员pbData和dwDataLength是存放在结构体中的映像数据的缓冲区的指针和位组长度，之后的两个成员pbSig和dwSigLength是存放在当前结构体中的数字签名的缓冲区指针和以字节为单位的数字签名的长度。dwRecAddress、dwRecLength、dwRecCheck这3个成员是当前结构体中的映像数据所属记录的信息，其含义分别与BL_ IMAGE_TYPE_BIN类型映像的记录头文件信息完全相同。对于较小的记录，一个结构体可以包含一个记录，较大的记录则需要分割成多个结构体。对于需要分割记录数据的情况，dwRecAddress、dwRecLength、dwRecCheck这3个成员记录的数值仍然是原始的记录的属性信息，因为分割的单元是记录中的数据，而不是记录本身。bRandomSeed成员是16字节的随机数，在有数字签名的BL_IMAGE_TYPE_ SIGNED_BIN类型映像文件中，继4字节的映像目的物理储存起始地址和4字节的映像数据位组长度之后，被Boot Loader读出的就是这个随机种子，属于同一个映像文件的所有结构体都拥有一个相同的16字节随机种子数据。dwSequenceNumber成员是结构体的序号，在一个记录分属多个结构体的情况下使用。bFlags成员用于记录结构体的属性。

PACKETDATA结构体所组织的结构体仅存在结构体数据被完全下载到内存中之后，这个数据结构主要供Boot Loader校验其数字签名使用，而Boot Loader通过下载端口获取的结构体仍是一种结构型的数据块。包含一个完整的记录的结构体，都有一个9字节的结构体，分别是4字节的结构体数据长度、4字节的数字签名长度和1字节的结构体属性数据，结构体数据长度只包括记录中的有效数据，不包括12字节的记录（内存起始地址、长度和校验码）。结构体之后是记录头属性，然后是结构体数据（即记录的映像数据），最后是与结构体的数字签名长度对应的数字签名数据。对于一个记录分属多个结构体的情况，则结构稍微复杂一些，属于同一记录的第一个结构体在结构体头之前，还有一个块（chunk）头，也是9字节，前4个字节被忽略，中间4字节是当前记录所拥有的结构体数，用于指出后继还有多少结构体是属于这同一记录的，最后1字节是结构体属性信息。分属多个结构体的记录在每个结构体中其头文件信息都是相同的，都是针对整个记录的，被分割的只是记录中的数据，记录的头文件是不可分割的。

当一个结构体的全部资料都从下载端口读出以后，按照Manifest数据的要求被放置到适当的储存位置（RAM内存或Flash中），然后根据结构体数据的头文件信息，连同之前获取的16字节随机数种子一起填充到一个PACKETDATA结构体类型的数据中，最后调用SBL_ VerifyPacket函数对结构体数据执行认证，通过了认证的结构体的映像数据才会被Boot Loader接受，否则DownloadSignedBin函数返回FALSE表示下载映像失败。使用数字签名进行认证的原理和SBL_ VerifyPacket函数的算法都十分复杂，并且与Windows CE Boot Loader的开发实践关系不大，这里暂且略过，感兴趣的读者请自行研读SBL _VerifyPacket函数的原始程序代码。

相对于BL_IMAGE_TYPE_SIGNED_BIN类型来说，BL_IMAGE_TYPE SIGNED_NB0类型的映像文件要简单得多。因为.nb0文件是完全没有格式的，所以带数字签名的.nb0文件中只有

结构体，无须处理记录和记录的分割（块）。BL_IMAGE_TYPE_SIGNED_NB0类型映像文件中，在7字节的特征码“R000FFk\x0A”之后是16字节的随机数种子，然后是逐个装载映像数据的结构体。BL_IMAGE_TYPE_SIGNED_NB0类型的映像文件与不带数字签名的.bin文件结构十分相似，它的结构体就类似.bin文件的记录。

BL_IMAGE_TYPE_SIGNED_NB0类型映像文件中结构体的定义和认证方法与前述的BL_IMAGE_TYPE_SIGNED_BIN类型映像文件完全相同。

## 5.4 Windows CE的系统启动执行

### 5.4.1 OAL的功能

Windows Embedded CE 6.0之所以能够在不同的硬件平台上执行，都是OAL的功劳。Windows Embedded CE 6.0以同样的方式向下请求（访问）硬件完成某种工作，但不同的硬件有不同的工作方式，也必须有不同的用于读取的程序代码，OAL可以在二者之间建立一种关联，屏蔽不同硬件平台的差异，让操作系统内核程序代码尽量做到硬件无关性。OAL的硬件操作仅仅只是为了完成Windows Embedded CE 6.0操作系统在启动和执行过程中最基本的硬件操作要求。

从物理意义上讲，OAL是一些库文件，是对操作系统请求（访问）硬件的抽象，每一个功能就是一个抽象的库。当操作系统需要请求某个硬件的某个功能时，只需要读取这些库即可。从此可以看出，其实OAL也是操作系统的一部分，只不过它比内核更贴近硬件，与硬件的相关性更大。OAL中的库与内核库进行连接，就形成了Windows Embedded CE 6.0操作系统内核的可执行文件。

对于一个Windows Embedded CE 6.0内核映像，以下几个库是必需的。

- NK.lib：与特定处理器相关的程序代码，一般由微软公司提供。
- NkProf.lib：可选的，它是微软公司提供的处理器相关内核程序代码，用于性能监测，当Windows Embedded CE 6.0操作系统属性页配置属性的Build Options中的Enable proofing（IMGPROFILER=1）被勾选时，它才会被加入。
- KITL.lib：当KITL不是一定要存在时，它也是可选的。主要是提供内核KITL除错服务的支持，当在Windows Embedded CE 6.0操作系统属性页配置属性的Build Options中的Enable KITL（no IMGNOKITL=I）被勾选时，它才会被加入。
- FullLibC.lib：微软专为Windows Embedded CE设计的C执行时库。

由于选择的不同，将有可能产生以下三个结果。

- Kern.exe：该内核不包含除错子系统。
- Kernkitl.exe：内核包含KITL子系统。
- Kernkitlprof.exe：内核包含KITL和除错子系统。

实现一个OAL是整个基于Windows Embedded CE 6.0设备开发周期中非常复杂的工作之一，通常的做法并不是从头到尾地编写一个OAL，而是从类似的平台中复制一个OAL，再对其进行特性化的修改，以适应目标设备的硬件平台。

当完成Windows Embedded CE 6.0设备的操作系统的创建之后，OAL存在于Windows

Embedded CE 6.0操作系统执行时映像中。

不过有些Windows Embedded CE 6.0设备启动后，并不是执行Boot Loader，而是直接加载操作系统映像，那硬件的初始化工作由哪部分来完成呢?这就得靠操作系统中的OAL层来完成，OAL部分程序代码与Boot Loader是共享的，这部分共享的程序代码就是完成硬件的初始化的工作中的一部分。当然，OAL层进行硬件初始化工作的前提是没有Boot Loader来进行硬件初始化，如果Boot Loader存在OAL就没有必要进行硬件初始化的工作了。另外，Boot Loader与OAL层的接口访问程序代码也是可以共享的。

PQOAL（the Production Quality of OAL）使得开发一个OAL变得更简单，周期上也更短。PQOAL以程序代码库和目录结构的重复使用、中心化的配置文件、一致的处理器家族和硬件平台等方式来让OAL更加组件化。通过它提供的处理器和硬件平台框架可以方便地开发一个最基本的Windows Embedded CE 6.0内核（Tiny Kernel Image）。

Windows Embedded CE 6.0的PQOAL有以下几个方面的重要改进。

- 常用的特定于处理器的组件集；
- OAL软件组件；
- 标准目标结构；
- BSP开发的惯例和约定。

在Windows Embedded CE 6.0的BSP中，有部分程序代码被存放在%_WINCEROOT%\Platform\<Hardware Platform Name>目录中，其实，这个目录只包含BSP很少的一部分程序代码，还有一部分程序代码被存放在%_WINCEROOT%\Platform\Common目录中，一般情况下并不需要修改Common目录中的文件。

## 5.4.2 OAL的构成

以上提到PQOAL的改进中，有一点是BSP开发的惯例和约定，惯例和约定主要是指PQOAL和BSP的配置标头文件。之所以称之为惯例和约定，是因为这些并不是非要遵守的，多是前人的一些经验的总结，遵循这些惯例和约定会给开发工作带来诸多的便利之处，在多人协作的情况下，更能给交流带来方便，对于系统维护和升级，带来的方便是不言而喻的。约定俗成的文件名如表5-3所示。

**表5-3 约定的配置标头文件名及描述**

| 标头文件名 | 描 述 |
|---|---|
| Args.h | OAL与Boot Loader之间共享的变量和数据结构 |
| Boot.h | 为Boot Loader定义标准库层次结构的组件标头文件 |
| Boot_efg.h | 基于Boot Loader的内存地址映像 |
| Bsp.h | 包含所有其他的库档，开发人员可以只在自己的程序代码中包含此标头档 |
| Bsp_base_reg_cfg.h | 板级缓存器地址定义 |
| Bsp_cfg.h | 包含基本的BSP硬件常量，比如CPU频率、SYSINTR定义 |
| Image_cfg.h | 基于映像的内在地址映像 |
| Ioctl_cfg.h | 目标设备硬件平台IOCTL用到的常量 |

（续）

| 标头文件名 | 描　述 |
|---|---|
| Ioctl_tab.h | IOCTL要处理的函数表 |
| Kitl_cfg.h | 与KITL相关的配置信息，如其支持的网卡列表 |
| Oemaddrtab_cfg.h | 它只存在于ARM平台中，定义OEMAddressTable，它是物理地址与虚拟地址的映像表 |

以上这些标头档会被%_WINCEROOT\Platform\<Hardware Platfom Name>\ src\Kernel\OAL目录中的一些C源文件用到，如表5-4所示。

表5-4　C源文件名及其描述

| 源文件名 | 描　述 |
|---|---|
| Args.c | 含有用于数据处理的参数或者Boot参数 |
| Debug.c | 含有处理除错的程序代码，比如串行口的除错 |
| Init.c | 含有OAL和OEMInit函数的实现程序代码 |
| Intr.c | 含有板级中断处理的程序代码 |
| Ioctl.c | 含有目标设备硬件平台的IOCTL实现的程序代码 |
| Kitl.c | 含有用于KITL的硬件平台初始化程序代码 |

OAL有很多库，不同的库实现不同的功能，这些库加在一起组成了一个OAL，Windows Embedded CE 6.0的操作系统内核再去利用库中的函数来实现硬件资源的请求（访问）。OAL的库结构如图5-4所示。

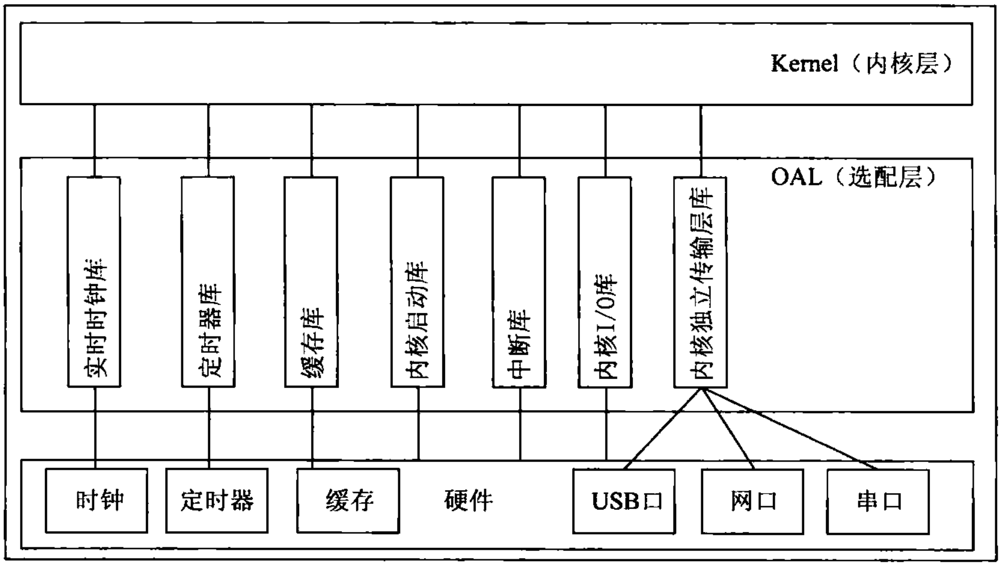



图5-4　OAL的库结构

## 5.4.3 OAL的启动

微软OAL的启动流程如图5-5所示。

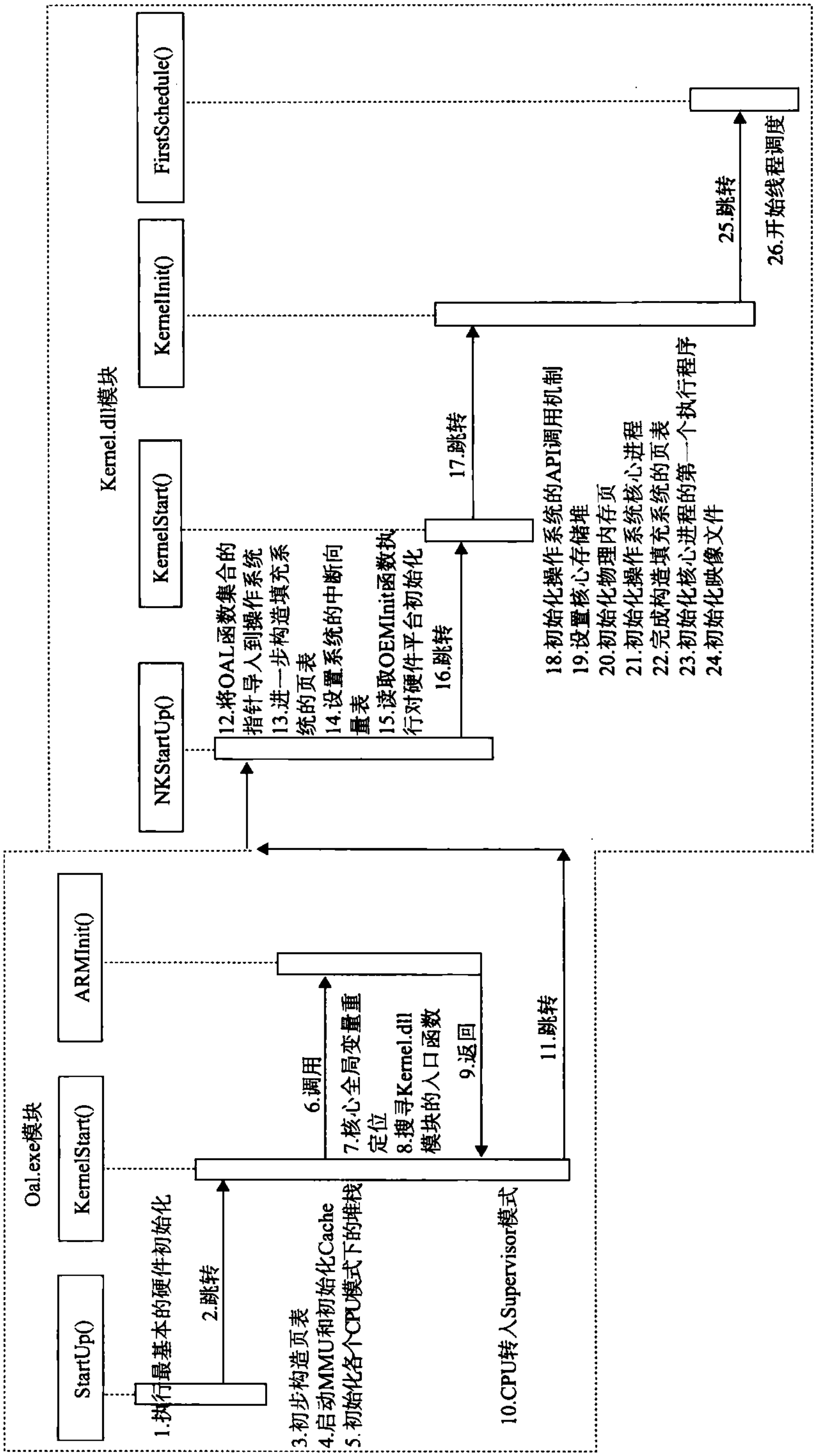



图5-5 Windows CE的初始化启动流程图

### 1. Startup

Startup一般用汇编语言书写，OAL与Boot Loader共享此段程序代码。Startup主要对硬件进行初始化、处理器模式切换、关闭MMU、并检测是否重新启动、中断、内存、时钟、电源等。（驱动程序之后还可以再调整和自己相关的GPIO/IRQ/Timer等）。

如果基于Windows CE的嵌入式系统软件不使用Boot Loader，则系统开机或重新启动后，直接执行的是操作系统的二进制代码，Windows CE的OAL模块的Start Up函数是第一个被嵌入式CPU执行的函数。尽管使用Boot Loader对嵌入式目标产品的开发和除错有很多好处，但是Boot Loader并不是嵌入式软件系统必不可少的组成部分。严格说来，Start Up函数需要OEM用户负责开发实现，应该归类为OAL模块的OEM函数。DMA-6410XP的OAL模块的Start Up函数在汇编源文件%_WINCEROOT%\PLATFORM\DMA6410XP\Src\Oal\Oallibstartup.s中实现（此内容在随机附赠的光盘中可以找到）。Boot Loader的第一个执行函数也是汇编语言的Start Up函数。Windows CE操作系统习惯将嵌入式系统开机执行的第一个函数命名为Start Up，因而造成了这里的重复名称的函数。但是同Boot Loader的Start Up函数一样，OAL模块的这个第一执行函数也可以使用其他的函数名而不是Start Up。以下是从DMA-6410XP的OAL模块的sources项目文件撷取的部分内容：

```
RELEASETYPE=PLATFORM
TARGETNAME=oal
TARGETTYPE=LIBRARY
SYNCHRONIZE_DRAIN = 1
NOMIPS16CODE=1
EXEENTRY=StartUp
```

用户可以通过更改其中的EXEENTRY项的数值，而使用其他的函数名代替Start Up。

以上其他选项解释如下。

- TARGETNAME和TARGETTYPE项的数值表示OAL的源代码最终编译链接生成一个名为oal.exe的二进制可执行文件。TARGETTYPE项可能的数值除PROGRAM表示的.exe文件外，还可以是LIBRARY和DYNLINK，后两者分别代表.lib静态库文件和.dll动态链接库文件。
- SYNCHRONIZE_DRAIN项数值为1表示build.exe对这个可执行文件的编译链接发生在所有被它依赖的库文件（.lib或.dll文件）编译链接完成之后。如果用户的开发计算机是单CPU的就不存在这个问题，build.exe按照项目所在目录的上层目录中的dirs文件所记录的顺序逐一编译链接每一个子目录中的项目。但是当用来执行Windows CE开发的开发计算机是多CPU时，多个编译链接的任务可以同时执行，此时就需要这个SYNCHRONIZE_DRAIN项目项取数值1来起作用了。
- RELEASETYPE项数值为PLATFORM表示编译链接最终生成的oal.exe可执行文件的存放在%_WINCEROOT%\目录下的各OS Design专属的子目录中。
- 在使用ARM架构CPU的目标平台上，NOMIPS16CODE项目项没有实际意义，仅为了向下兼容而存在。

OAL的Start Up函数的功能与Boot Loader中的同名函数相似，甚至比它更简单。

Boot Loader的Start Up函数要完成CPU芯片级的初始化、复制自身到RAM和设置页表，并且启用MMU这3项任务，而OAL的Start Up函数只需完成第一项，即嵌入式CPU芯片的初始化。

Windows CE操作系统的映像通过Boot Loader从本地读取，或是从网络下载到RAM中，或在ROM型的储存设备中以XIP的方式执行，一般不需要在OAL的Start Up函数加入复制自身的程序代码。但是如果特定的嵌入式产品确实有这个需求，在技术上是完全可行的，开发者可以比照Boot Loader中的相关程序代码进行设计实现，不过这仅适用于操作系统映像较小的情况。如果它的数据量达到一定的规模，会严重影响操作系统的启动速度，以致使用者不能忍受。Windows CE的Boot Loader通常大不过几百KB，不必考虑这个问题，但是复制一个几十MB的操作系统映像，就不得不考虑所需时间的影响了。

由于Windows CE操作系统所使用的虚拟内存较Boot Loader要复杂得多，导致仅靠一个汇编语言的Start Up函数不可能全部完成设置页表，并且启用MMU单元的任务。于是OAL的Start Up函数要做的工作就只剩下对CPU芯片的SOC级的初始化这一项了。前面介绍Boot Loader开发时讲过，对于开发者微软所建议的在Windows CE的Boot Loader应实现的功能与特性中，有这样一项，就是同属于BSP组成部分的Boot Loader和OAL模块应该尽最大限制地共享对硬件初始化的程序代码。在Boot Loader的主要任务已接近全部完成，OEMLauch函数在执行跳转动作之前，先禁用了MMU，在一定程度上也是为了达到这一目的。很明显，OAL的Start Up函数可以，并且应该利用Boot Loader的同名函数中的对CPU初始化的程序代码，主要包括：清空TLB和Cache、关中断、配置PLL、设置内存控制器等。与Boot Loader相似，OAL除了在Start Up函数中进行对CPU芯片的初始化外，还有一个与OEMPlatformInit函数对应的板级初始化函数OEMInit。

OAL的Start Up函数在完成了对CPU的初始化任务以后，跳转到下一个汇编函数KernelStart。

### 2. OALStartUp

OALStartUp传递OEMAddress Table到KernelStart。

### 3. KernelStart函数

继OAL的Start Up函数之后，被目标嵌入式系统执行的是KernelStart函数，这也是一个汇编语言函数，但是与Start Up函数不同，它的原代码由微软提供，而不是需要用户开发的OEM程序代码。KernelStart函数的实现原代码在%_WINCEROOT%\PRIVATE\WINCEOS\COREOS\NK\LDR\ ARM\armstart.s汇编源文件中。在\WINCE600\PRIVATE\WINCEOS\ COREOS\NK\LDR目录下的sources项目文件指示编译系统使用其子目录下的源文件编译产生一个名为nkldr.1ib的静态库文件：

```
TARGETNAME=nkldr
WINCECPU=1
TARGETTYPE=LIBRARY
CDEFINES=$(CDEFINES) -DIN_KERNEL -DNK_LOADER
```

```
...
ARM_SOURCES=      \
    arm\arminit.c    \
    arm\armstart.s
```

nkldr.Lib正是链接产生Windows CE的OAL模块中可执行文件oal.exe所需的静态库文件之一。

KernelStart函数的主要任务是为Windows CE操作系统设置页表，并且启用MMU。但是仅靠这一个汇编语言函数不可能设置全部的页表，KernelStart函数只构造操作系统所需的全部页表的一部分，但是这一部分就已经足够让ARM CPU的MMU硬件单元和Windows CE操作系统的虚拟内存软件机制正确有效地执行起来。虚拟内存机制，也称为虚拟内存技术，就是利用计算机系统的MMU硬件单元，将CPU发出的虚拟内存地址转换成实际的系统物理地址的机制。MMU单元实现虚实地址转换的依据是用户根据自己的需求建构并设置的页表。至少有以下3个好处促使嵌入式CPU的设计者和嵌入式系统的开发者采用虚拟内存技术。

- 使用虚拟内存可以根据用户的需要重新分配内存地址空间。系统物理储存地址空间的安排分配通常取决于CPU和板级硬件电路的设计，软件开发者几乎不可能改变，这有时会给系统软件的设计带来麻烦。以ARM CPU为例，当发生外设中断请求时，程序执行发生跳转，CPU的PC缓存器要被赋值为0x00000018。假如这个物理地址被ROM类型的储存设备占据，则中断处理程序必须在ROM内存中以XIP的方式执行，这会大大降低对外设中断的响应速度，也给中断处理程序的设计造成困扰。有了虚拟内存技术，系统软件开发者在建构页表时，将虚拟内存地址0x00000000起始区域映像至RAM内存的某一块储存空间，当系统发生中断跳转时，MMU单元将0x00000018虚拟地址映像至RAM内存中，中断处理程序将在RAM中执行，问题也就迎刃而解了。
- 使用虚拟内存技术可以避免硬件物理地址空间的变化对软件的影响。当因硬件设计发生改变而导致系统物理地址发生变化时，假如没有虚拟内存技术，系统软件设计者将不得不全面修改程序代码中有关的部分。这对于操作系统这样的庞大又复杂的系统软件来说，工作量将会大得不可想象。但是有了虚拟内存，软件开发者只需调整页表，保证虚拟内存地址空间改变尽可能小，就可以将工作量降低到最小。
- 使用虚拟内存技术可以对储存空间引入保护机制。在为MMU构造的用于虚实地址转换的页表中，不仅记录了虚实地址映像信息，还记录了所对应的储存空间的读、写、可执行等储存属性。这些属性由MMU硬件定义并且负责遵守执行，页表项中的属性具体数据内容由系统软件开发者根据需要填写。比如，在Windows Embedded CE 6.0操作系统中，虚拟内存地址空间的高2GB（0x80000000～0xFFFFFFFF）属于操作系统内核专用，用户模式行程完全无权访问，则开发者应将高2GB虚拟内存地址所对应的页表项的储存属性设为内核模式可擦写、用户模式无权访问。

表5-5所示为基于ARM架构CPU的Windows Embedded CE 6.0操作系统所使用的虚拟内存地址空间使用分配情况。

表5-5 虚拟内存地址空间使用情况

| 地址范围 | | 容量大小 | 用途描述 |
|---|---|---|---|
| 低2GB，用户模式 | 0x00000000 ~ 0x000010000 | 64KB | 为安全性预留 |
| | 0x00100000 ~ 0x3FFFFFFF | 64KB ~ 1GB | 用户模式可动态分配的虚拟内存 |
| | 0x40000000 ~ 0x5FFFFFFF | 512MB | ROM DLL虚拟内存空间 |
| | 0x60000000 ~ 0x6FFFFFFF | 256MB | RAM—backed memory—mappings |
| | 0x70000000 ~ 0x7FFFFFFF | 255MB | 共享堆 |
| | 0x7FF00000 ~ 0x7FFFFFFF | 1MB | 为安全性预留 |
| 高2GB，内核模式 | 0x80000000 ~ 0x9FFFFFFF | 512MB | Cacheable静态映像区 |
| | 0xA0000000 ~ 0XBfffffff | 512MB | Uncacheable静态映像区 |
| | 0xC0000000 ~ 0xC7FFFFFF | 128MB | Kernel XIP DLL |
| | 0xC8000000 ~ 0xCFFFFFFF | 128MB | 对象储存 |
| | 0xD0000000 ~ 0xEFFFFFFF | 512MB | 内核模式可分配虚拟内存 |
| | 0xF0000000 ~ 0xFFFFFFFF | 256MB | 与CPU架构相关的虚拟内存 |

Windows CE操作系统在基于ARM架构CPU的嵌入式系统上执行时CPU的MMU单元采用二级页表结构。按照微软提供的通用原始程序代码的规范命名，一级页表称为页目录（Page Directory），二级页表称为页表（Page Table）。但是为了不至于引起误会，在下文不使用“页目录”和“页表”这两个名词，仍然使用更容易理解的“一级页表”和“二级页表”两个称谓。

在KernelStart函数所在的汇编源文件armstart.s中，首先定义了一个名为.KDATA的数据区域，这个区域预留了24KB（6个页面）的物理储存空间，其中前4页用作一级页表的储存空间，第5页用于存放异常向量和CPU特权级的堆栈，第6页的前3KB放置内核堆栈和KDataStruct结构体，最后1KB用作虚拟地址0xFFF00000段（Section）的二级页表。

```
; KDATA数据区用于存储上述结构的物理内存
        AREA |.KDATA|,DATA,NOINIT
KDataArea
PTs     % 0x4000 ; 一级页表存储空间
ExceptionVectors
        % 0x0400 ; 异常向量存储空间
        % 0x0400 ; 中断堆栈存储空间
        % 0x0100 ; FIQ堆栈存储空间
        % 0x0700 ; 中止堆栈存储空间
KPage   % 0x0c00 ; 内核堆栈和KDataStruct结构体
HighPT  % 0x0400 ; 二级页表虚拟地址0xFFF00000段
KDEnd   % 0
```

KernelStart函数首先为起始地址为0xFFF00000的虚拟内存段设置二级页表，这个二级页表储存在以High PT为起始地址的1KB物理RAM空间内。每一个二级页表项对应于4KB的虚拟内存空间，每个表项本身占用4字节物理储存空间，1KB的二级页表总共可以容纳256个表项，则整个页表总共对应于1MB的虚拟储存空间，刚好是一个内存段（Section）的容量大小。这个起始地址为0xFFF00000的虚拟内存段在微软提供的通用原始程序代码中被称为ArmHigh区域。此二级页表的内容如表5-6所示。

表5-6 二级页表内容

| 起始虚拟内存地址 | 起始物理内存地址 | 属 性 |
|---|---|---|
| 0xFFFD0000 | PT（一级页表所在地） | 4KB小页面；内核模式可擦写；非可执行 |
| 0xFFFD1000 | PT+0x1000 | 4KB小页面；内核模式可擦写；非可执行 |
| 0xFFFD2000 | PT+0x2000 | 4KB小页面；内核模式可擦写：非可执行 |
| 0xFFFD3000 | PT+0x3000 | 4KB小页面；内核模式可擦写；非可执行 |
| 0xFFFD4000 | ExceptionVcctors（异常向量和堆栈所在） | 4KB小页面；内核模式可擦写 |
| 0xFFFFC000 | KData页 | 4KB小页面；内核模式可擦写；用户模式只读 |

KernelStart函数执行的第二步骤是为起始地址为0xFFF00000的虚拟内存段设置一个一级页表项。一级页表的每一表项对应1MB的虚拟内存空间，即一个段。一级页表本身共占用16KB的物理内存空间，每一页表项占用4B，整个一级页表即可映像全部的4GB虚拟内存地址空间。之前设置的二级页表所对应的起始虚拟地址是0xFFF00000，所以它位于一级页表的最后一项。下面的程序代码将储存该二级页表的物理内存的起始物理地址写入一级页表的最后一项；并且将属性设置为PTL1_2Y_TABLE，这个属性表示这里的一级页表项对应着一个二级页表，而不再是直接映像为1MB的物理内存段。

```
orr r0,r4, #PTL1_2Y_TABLE          ; (r0) = 高端存储的一级页表PTE
add r1,r10, #0x4000
str r0,[r1, #-4]                   ; 存储一级页表PTE最后一页
```

ARM CPU的MMU单元将虚拟内存地址转换成系统物理地址的方法如下：首先将虚拟地址的高12位作为查找一级页表的索引。如果对应的一级页表项的属性不是PTL1_2Y_TABLE，则取虚拟地址的低20位作为对应的物理地址的偏移值，加上一级页表项中记录的物理地址起始值，即得出实际的系统物理地址；反之，如果一级页表项的属性包含有PTL1_2Y_TABLE，则虚拟地址的低20位还不是物理地址的偏移值，还要根据这20位数据中的高8位（由低到高第12至第19位）查找一个二级页表。一级页表项中的地址数据就是待查找的二级页表的起始物理储存地址，查找二级页表得到的物理地址加上虚拟地址的最低12位才得到最终的系统物理地址。

KernelStart函数接下来的第3个执行步骤要做的事情是填充一级页表的0x80000000 ~ 0x9FFFFFFF虚拟内存区域。这部分虚拟内存将要容纳目标嵌入式系统的全部的物理地址空间，包括各种类型的CPU可直接寻址的物理储存设备和各种外设的控制与状态缓存器、内置数据缓冲区和IO端口所占用的物理地址。这部分一级页表项填充的依据是g_oalAddressTable数组。这部分的页表项属性均设置为PTL1_SECTIONIPTL1_KRW，PTL1_SECTION属性与PTL1_2Y_TABLE刚好相反，表示一个一级页表项直接映像一个1MB的物理存储段，PTLI_KRW属性则表示该虚拟内存段在内核模式下可擦写。KernelStart函数仅填充一级页表的0x80000000 ~ 0x9FFFFFFF虚拟地址空间部分，这部分本来是被Windows CE操作系统定义为cachable的静态映像区间，但是暂时还没有为这些一级页表项设置cachable属性。KernelStart函数可以启用MMU了，尽管到目前为止还只填充了全部页表的一部分，但是已经足够使MMU单元正常工作了。

KernelStart函数为Windows CE操作系统设置页表的依据是g_oalAddress Table数组，所以

KernelStart函数以g_oalAddressTable数组数据的存储起始地址为函数参数。OAL模块的程序执行流程是由Start Up函数跳转到KernelStart函数的，也可以说是OAL的Start Up函数读取KernelStart函数，所以Start Up函数需要为KernelStart函数准备参数。记录g_oalAddressTable数组数据的是oemaddrtab_cfg.inc标头文件，它被包含在Start Up函数所在的%_WINCEROOT%\PLATFORM\DMA6410XP\SRC\OAL\OALLIB\startup.s源文件中，但是Start Up函数不能直接把g_oalAddressTable符号的数据赋值给r0作为KernelStart函数的参数。因为在这一函数读取发生时，MMU单元还没有启用，CPU访问外部物理存储仍然使用的是系统物理地址，而g_oalAddress Table符号记录的是g_oalAddressTable数组的起始存储位置在0x80000000 ~ 0x9FFFFFFF范围内的虚拟内存地址。之所以g_oalAddress Table符号记录的是虚拟内存地址，而不是物理地址，并且落在了0x80000000 ~ 0x9 FFFFFF虚拟内存地址范围内，是因为在汇编原始程序代码中使用的是用来表达程序地址的字符串在程序链接的阶段要接受地址重新定位的处理，而地址重新定位的依据是DMA6410XP BSP的项目%_WINCEROOT%\PLATFORM\DMA6410XP\ FILES\config.bib（此BSP为随机附赠的光盘中的内容），其中对系统物理存储的使用分配作出安排所使用的全部是0x80000000 ~ 0x9FFFFFFF范围内的虚拟内存地址。

鉴于以上所述的原因，OAL的Start Up函数使用以下的程序代码为KernelStart函数准备参数：

```
;执行内核启动
   add r0,pc, #g_oalAddressTable - (. + 8)
   bl  KernelStart        ; 调用Windows CE内核
```

以上程序代码中的“.”符号记录的是它所在的汇编指令的储存地址，同字符串所代表的地址一样，它也要经过链接阶段的地址重新定位处理，最终转换成在0x80000000 ~ 0x9FFFFFF范围内所对应的虚拟内存地址。而在上面的add指令被执行时，MMU单元还没有启用，PC缓存器中装载的是执行指令的物理存储地址，再考虑到S3C6410的CPU所使用的3级流水线结构，则PC缓存器中的数据就是当前正在执行的指令的物理储存地址加8。所以如果把上面的add指令所执行的运算改写成下面的形式，读者就能够明白add指令执行完成之后r0缓存器中最终存放的数据了，就是g_oalAddressTable数组的起始物理储存地址：

```
#g_oalAddressTable_[.-(pc-8)] -> r0
```

所有的准备工作都做好了，KernelStart函数剩下的工作主要就是启用CPU的MMU单元，从这里开始Windows CE操作系统的程序执行流程就进入了使用虚拟内存阶段，而不是原始的物理存储地址。其后，在KernelStart函数中还要为ABORT、IRQ、FIQ、UNDEF、Supervisor这5个CPU的特权级状态设置堆栈。

前述工作完成以后，KernelStart函数跳转到ARMInit函数。这是一个C语言函数，但是这个函数执行完成以后仍然会回到KernelStart函数中来，它只能算是KernelStart函数用高级程序语言编写的部分程序代码。ARMInit函数向KernelStart函数返回kernel.dll动态链接库的入口点函数的指针，最后KernelStart跳转到kernel.dll的入口点函数继续执行。

#### 4. ARMInit

ARMInit可读取KernelRelocate（pTOC），可走出纯ROM，定位DLL和变量。这是Windows CE操作系统启动执行流程中的第一个高级语言函数，它的原始程序代码在源文

件%WINCEROOT%\PRIVATE\WINCEOS\COREOS\ NK\LDR\A RM\ARMInit.c中。严格地说，ARMInit函数并不构成系统启动执行流程的独立组成部分，它只是被KernelStart函数读取，并且还要返回到KernelStart函数中来。

ARMInit函数的主要任务有两个：一是读取KernelRelocate函数进行内核整体变量重新定位；二是读取FindKernelEntry函数查询动态链接库kernel.dll的入口点函数地址。读者应该还记得，前面的Boot Loader的程序代码也要做整体变量重新定位的工作，它和这里的内核整体变量重新定位原理是完全一样的，只是操作的对象不同。ARMInit函数完成这两项任务都要使用Windows CE操作系统映像.bin文件中的TOC信息。

5. NKStartup

NKStartup可初始化KDataStruct *pKData内核整体变量、定位OEMGLOBAL结构体、ARMSetup、OEMInitDebugSerial、OEMInit。

NKStartup函数在%WINCEROOT%\PRIVATE\WINCEOS\COREOS\NK\ KERNEL\ARM\mdarm.c源文件中实现。程序执行到了这里，Windows CE操作系统的启动执行流程已经从oal.exe模块转移到了kernel.dll模块。NKStartup函数的说明已经比较详细地列举出了这个函数所需完成的任务。

1）由NK Loader传递过来的函数参数（即KdataStruct结构体数据）中获取出供自身执行所需的数据信息。

2）找出用户OAL函数的入口点。

3）填充虚拟内存的uncachable静态映像区域0xA000000 ~ 0xBFFFFFFF部分，设置中断向量表。

4）如果系统启用了KITL除错功能，则加载KITL模块。

5）读取用户OAL函数执行对目标平台板级硬件的初始化任务。

以上各项任务中直接在NKStartup函数中完成的，除第一项从参数中获取所需数据外，只有第二项获取用户OAL函数的入口点，完成这项任务在程序代码中的实现，就是为整体变量g_pOemGlobal赋值：

```
//找到OAL入口
pfnInitGlob = (PFN_OEMInitGlobals)pKData->dwOEMInitGlobalsAddr;
//检查这里,如果OAL入口存在,则继续
g_DOemGloba~ = pfnln~tGlob(g~NKGl~bal);
```

赋值过程中会读取函数（指针）pKData→dwOEMInitGlobalsAddr，很容易查出，它是在ARMInit函数中赋值的：

```
pKData->dwOEMInitGlobalsAddr = (DWORD)OEMInitGlobals;
```

OEMInitGlobals函数在源文件%_WINCEROOT%\PUBLIC\COMMON\ OAK\INC\oemglobal.h中实现。在原始程序代码的注释里称这个函数为OAL的入口。准确地说，应称它为OAL与操作系统的衔接点。这个函数的实现相当简单，主要是把OEMGLOBAL型静态整体变量OemGlobal的地址赋值给同类型的整体变量指针g_pOemGlobal，剩下的部分主要在静态整体变量OemGlobal及其类型OEMGLOBAL结构体上做。在标头文件%_WINCEROOT%PUBLIC\COMMON\OAK\INC\oemglobal.h中有OEMGLOBAL结构体的完整定义，它的大多数成员都是函数指针类型。微软深得对象导向程序之精妙，可惜C语言不支持成员函数，只好用函数

指针类型的数据成员代替。实际上C++语言的成员函数最终也是依靠函数指针来实现的。在源文件oemglobal.c中定义了OemGlobal整体变量，并且同时为它的各个成员赋值，在被用来给OemGlobal的各个函数指针型的数据成员赋值的函数中，凡是以OEM为函数名前缀的都是必须由BSP开发者负责开发实现的OAL函数，所有的用户实现的OAL函数都在这里提交到Windows CE操作系统。OEMGLOBAL结构体用做OAL函数和其他操作系统启动信息的相关数据结构。

在后面的程序代码中，将会看到这些OAL函数是如何在嵌入式的软件系统中起作用的。以初始化除错串行端口的OEMInitDebugSerial函数为例，被内核原始程序代码读取的OEMInitDebugSerial函数是这样在源文件%_WINCEROOT%\PRIVATE\WINCEOS\COREOS\NK\OEMSTUB\oemstub.c中实现的：

```
void OEMInitDebugSerial(void)
{
    g_pOemGlobal->pfnInitDebugSerial ();
}
```

这个pfnInitDebugSerial成员在整体变量OemGlobal中被赋值为OEMInitDebugSerial，这正是用户在OAL中实现的用来初始化除错串行端口的OEM函数，其他OAL函数的使用方式都与此类似。

填充虚拟内存的uncachable静态映像区域和设置中断向量表的任务由NKStartup函数通过读取ARMSetup函数来完成，下一小节将专门介绍这个函数的功能和实现方法。

在NKStartup函数中还有一件十分重要的工作——读取OAL函数OEMInit执行对板级硬件的初始化。Start Up函数也进行初始化，但是仅限于CPU芯片级。OAL的OEMInit函数的作用与Boot Loader的OEMPlatformInit函数相似。DMA6410XP的OEMInit函数所完成的初始化任务主要有：初始化cache整体变量、初始化LCD显示、初始化中断。与BootLoader的OEMPlafformInit函数相比，OEMInit函数主要是多了初始化中断和可能的初始化KITL的任务。Boot Loader以轮询的方式操作外部设备，而Windows CE操作系统需要以中断的方式访问外部设备，初始化中断处理系统成为操作系统启动阶段的初始化任务的必需的组成部分。初始化KITL的任务之所以是可能的，是因为不是所有的基于Windows CE的嵌入式系统都要启用KITL除错功能，通常即使是同一目标嵌入式系统，也只会在产品的软件开发除错阶段启用KITL。

NKStartup函数在完成自己功能范围之内的所有任务之后，跳转到第二个KernelStart函数去执行，这也是一个汇编语言的函数。由于历史遗留的问题，这里又出现了重复名称的函数。不过没有关系，这一个KernelStart汇编函数与NKStartup函数同属一个静态库及同一个kernel.dll模块，而上一个KernelStart函数所属的nkldr.lib库被链接进了另一个Windows CE系统模块oal.exe。

### 6. KernelStart

第二个KernelStart函数在%_WINCEROOT%\PRIVATE\WINCEOS\ COREOS\NK\KERNEL\ARM\armtrap.s汇编源文件中实现。尽管后面还有部分的汇编程序代码，还有大量的初始化工作要做，但是到了这里，Windows CE操作系统的启动程序已经基本摆脱了目标平台硬件问题的干扰，进入了单纯的软件处理部分。

KernelStart函数首先读取KernelInit函数执行一系列复杂的软件初始化工作。KernelInit函数在%_WINCEROOT%\PRIVATE\WINCEOS\COREOS\ NK\KERNEL\nkinit.c源文件中实现，其实

现原始程序代码相当简单，具体如下：

```
VOid KernelInit(void)
(
#ifdef DEBUG
  g_pNKGiobal—>pfnWriteDebugString(TEXT("Windows CE KernelInit\r\n"));
#endif
    APICallInit();
    HeapInit();
    InitMemoryPool();
    PROCInit();
    VMInit(g_pprcNK);
    THRDInit();
        MapfileInit();
#ifdef DEBUG
  g_pNKGiobal—>pfnWriteDebugString(TEXT("Scheduling the first thread.\r\n"));
#endif
```

KernelInit函数读取APICallInit函数设置Windows CE的API机制；读取HeapInit函数设置内核堆内存；读取InitMemoryPool函数初始化物理内存页；读取PROCInit函数初始化操作系统内核行程；读取VMInit函数为内核行程设置虚拟内存的静态映像区域和ArmHigh区域以外的部分；读取THRDInit函数启动内核行程的第一个执行绪；读取MapfileInit函数初始化映像。

7. FirstSchedule

打开中断，多任务系统开始运转。在这一系列烦琐的系统初始化工作全部完成以后，Windows CE的启动程序即进入行程除错程序代码。至此，Windows CE这个庞大而复杂的软件机器就开始运转起来了。

## 5.5 DMA-6410XP中Windows CE OAL开发

这里的OAL指的是Windows CE的BSP的3个组成部分之一的用户OAL函数集合，而不是Windows CE操作系统的OAL模块。OAL是OEM Adaptation Layer的缩写，它是介于目标设备硬件和Windows CE操作系统内核之间的用户程序代码逻辑层。称它为用户程序代码逻辑层包括两层意思：首先是OAL主要由OEM用户根据目标设备硬件设计的特点而编写的程序代码，不同的目标硬件设计就有不同的OAL；其次是OAL本身是由一系列用户开发实现的OEM函数组成，被Windows CE的操作系统内核中间层程序代码读取以实现自身功能，在操作系统执行过程中不构成可执行模块实体。

我们常常将Windows CE的Boot Loader、OAL和设备驱动程序相提并论，这三者共同组成Windows CE操作系统的板支持包BSP（Board Support Package）。但是Boot Loader有自身完整的程序执行流程，而OAL和设备驱动程序只是一些供操作系统在执行过程中读取的函数的集合，从这个意义上来说，OAL与设备驱动程序有更多的相似之处。只不过Windows CE的设备驱动程序一般只是用来驱动某一特定外设的，系统中缺少某一驱动程序只会造成与之对应的外设不能正常工作。而OAL的主要工作是初始化和驱动供Windows CE操作系统内核启动执行所需的内核硬件如MMU、Cache、系统时钟、储存控制器、异常处理机制等，如果系统中缺少了

OAL，将造成操作系统无法启动执行。所以可以把OAL理解成是Windows CE操作系统基于硬件平台的内核硬件的驱动程序、所有设备驱动程序中的最大者。

## 5.5.1 OEMGLOBAL结构体及其整体变量

前述kernel.dll模块的入口点函数NKStartup的所需完成的任务之一是找出用户OAL函数的入口点。用户OAL函数，就是需要OEM用户负责开发实现的供OAL模块读取的功能函数。这些函数的性质和在OAL模块中所起的作用，与Boot Loader的OEM函数是相似的，并且它们也是以“OEM”这3个字符作为函数名的前缀，故称之为OAL的OEM函数，在本书中也简称为OEM函数。OAL的OEM函数不是一个，而是一系列函数，用户OAL函数的入口点，意思是Windows CE操作系统的启动程序可以通过读取这一个入口点函数而获取其余全部的OAL的OEM函数。在Windows CE操作系统中，这个入口点函数被命名为OEMInitGlobals。

虽然OEMInitGlobals函数也以OEM为函数名前缀，但是这并不是一个需要OEM用户开发实现的函数，它的实现原始程序代码在源文件%_WINCEROOT%\PRIVATE\WINCEOS\COREOS\NK\OEMMAIN\oemglobal.c中。这个目录下源文件被编译进了静态库oemmain.lib中，而这个静态库又是DMA-6410XP的OAL模块可执行文件oal.exe的组成部分。以下是从DMA-6410XP的OAL模块的sources项目文件中撷取的部分：

```
TARGETLIBS= \
        $(_COMMONOAKROOT)\Iib\$(_CPUDEPPATH)\oemmain-lib \
        $(_COMMONOAKROOT)\Iib\$(_CPUDEPPATH)\nkstub.lib \
        $(_COMMONOAKROOT)\Iib\$(_CPUDEPPATH)\nkldr.lib \
$(_TARGETPLATROOT)\Iib\$(_CPUDEPPATH)\oaI-Iib \
...
```

对基于ARM架构的CPU的嵌入式目标系统来说，OEMInitGlobals函数的执行原始程序代码可精简表示为：

```
POEMGLOBAL g_pOemGlobal = &OemGlobal;
...
POEMGLOBAL OEMInitGlobals(PNKGLOBAL pNKGlobal)
{
        g_pNKGiobal = pNKGlobal;
        g_pOemGlobal->dwARMCacheMode = OEMARMCacheMode ();
        return g_pOemGlobal;
}
```

这里真正起作用的是OEMGLOBAL结构体类型的静态整体变量OemGlobal，OEMInitGlobals函数的主要功能就是把它的地址指针返回给函数的读取者。

前面提过，OEMGLOBAL结构体是OAL模块的用户OEM函数的容器数据结构，它的成员多数都是函数指针类型，这些函数指针的类型定义实际上就是各OEM函数的函数接口定义。但是Windows CE操作系统内核并不是直接读取的记录在OemGlobal整体变量的各成员中的函数（指针）。以负责向系统的除错端口输出用户提示字符串的OEMWriteDebugString函数为例，系统内核的原始程序代码实际读取的是在源文件%WINCEROOT%\PRIVATE\WINCEOS\COREOS\NK\OEMSTUB\ oemstub.c中实现的OEMWriteDebugString函数，它的原始程序代码如下：

```
Void OEMWriteDebugString(unsigned short *str)
{
      g_~OemGlobal->pfnWriteDebugString(str);
}
```

在oemstub.c源文件中定义并实现的这类函数还不止OEMWriteDebugString函数这一个，它们都被编译进了静态库文件oemstub.1ib，而这个静态库是操作系统内核模块kernel.dll组成部分。Windows CE操作系统的内核原始程序代码通过读取oemstub.1ib的库函数间接地使用记录在OemGlobal整体变量的各成员中的用户OAL函数的功能。这些库函数的功能实际上代表了被它读取的OEMGLOBAL结构体的各成员所记录的用户实现OEM函数的功能定义。Windows CE OAL的开发者想要将自己开发实现的OEM函数提交给操作系统，其方法就是在OemGlobal整体变量的定义位置，或者随后的操作系统启动初始化程序代码中使用OEM函数的函数名（即函数指针）为相应功能的OemGlobal的函数指针成员赋值。

OEMGLOBAL结构体在标头文件%_WINCRROOT%\PUBLIC\COMMON\ OAK\ INC\ OEMGLOBAL.H中有完整定义。

## 5.5.2 OEM地址映像表

任何一个Windows CE的BSP的开发者都必须为它在原始程序代码中定义一个OEM地址映像表。不仅OAL模块要使用这个OEM地址映像表，Windows CE系统原始程序代码中，凡是涉及内存管理的地方都必须以它为最原始和最权威的依据，包括Boot Loader，如果要启用虚拟内存机制也必须使用OEM地址映像表来建构最基本的页表供MMU硬件单元使用。尽管如此，通常仍把建构OEM地址映像表看做Windows CE的OAL开发的一个组成部分。虽然OEM地址映像表的数据没有被纳入OEMGLOBAL结构体，但是其重要性绝不亚于OEMGLOBAL结构体的任何一个成员所代表的系统数据和OEM函数。DMA-6410XP的OEM位地址映像表在标头文件%_WINCEROOT%\PLATFORM\ DMA6410XP\Src\Inc\ oemaddrtab_cfg.inc中的定义如下：

```
g_oalAddressTable
    DCD   0x80000000,0x30000000,128           ; 128 MB DRAM BANK 6
    DCD   0x84000000+0x4000000,0x10000000,32  ; 32 MB SROM(SRAM/ROM)BANK 2
    DCD   0x86000000+0x4000000,0x18000000,32 ; 32 MB SROM(SRAM/ROM)BANK 3
    DCD   0x88000000+0x4000000,0x20000000,32 ; 32 MB SROM(SRAM/ROM)BANK 4
    DCD   0x8A000000+0x4000000,0x28000000,32 ; 32 MB SROM(SRAM/ROM)BANK 5
    DCD   0x95000000,0x08000000,32           ; 32 MB SROM(SRAM/ROM)BANK 1
    DCD     0x90800000,0x48000000,  1              ; SDRAM control register
    DCD     0x90900000,0x48800000,  1              ; EBI control register
    DCD     0x90A00000,0x49000000,  1              ; USB Host control register
    DCD     0x90B00000,0x49800000,  1              ; USB Device control register
    DCD     0x90C00000,0x4A000000,  1      ; Interrupt control register
    DCD     0x90D00000,0x4A800000,  1      ; HS-MMC control register
    DCD     0x90E00000,0x4B000000,  1      ; DMA control register
    DCD     0x90F00000,0x4B800000,  1      ; CF Card control register
    DCD     0x91000000,0x4C000000,  1      ; SYSCON register
    DCD     0x91100000,0x4C800000,  1      ; TFT-LCD control register
    DCD     0x91300000,0x4D800000,  1      ; Camera control register
    DCD     0x91400000,0x4E000000,  1      ; NAND flash control register
```

```
DCD     0x91500000,0x4E800000,  1       ; Matrix control register
DCD     0x91600000,0x4F000000,  1       ; SSMC control register
DCD     0x91700000,0x4F800000,  1       ; TIC control register
DCD     0x91800000,0x50000000,  1       ; UART control register
DCD     0x91C00000,0x51000000,  1       ; PWM control register
DCD     0x91D00000,0x52000000,  1       ; HS-SPI and SPI0 control register
DCD     0x91E00000,0x53000000,  1       ; WDT control register
DCD     0x91F00000,0x54000000,  1       ; IIC control register
DCD     0x92000000,0x55000000,  1       ; IIS control register
DCD     0x92100000,0x56000000,  1       ; I/O Port register
DCD     0x92200000,0x57000000,  1       ; RTC Port register
DCD     0x92300000,0x58000000,  1       ; TSADC Port register
DCD     0x92400000,0x59000000,  1       ; SPI Port register
DCD     0x92500000,0x5A000000,  1       ; SDI Port register
DCD     0x92600000,0x5B000000,  1       ; AC97 Port register
DCD     0x93000000,0x00000000,32        ; 32 MB SROM(SRAM/ROM)BANK 0
DCD     0x00000000,0x00000000,  0        ; end of table
```

OEM地址映像表是一个真实反映目标硬件平台的系统物理存储地址（包括物理存储设备和分配给各外设使用的物理地址空间）空间分布和物理存储与虚拟内存的地址映像关系的列表，它在系统原始程序代码中通常是设置为只读的。OEM地址映像表的每一表项由虚拟内存起始地址（Virtual Address）、起始物理地址（Physical Address）和以MB为单位的地址空间大小（Size）3个部分组成。比如上面列出的DMA-6410XP的g_oalAddressTable数组其第一表项即第一行所表达的信息是，从系统物理地址0x30000000开始处的128MB物理RAM内存空间被映像到虚拟内存的0x80000000 ~ 0x87FFFFFF地址范围。在高级语言原始程序代码中，通常使用OAL ADDRESS_TABLE结构体表示OEM地址映像表的表项，这个结构体的定义如下：

```
typedef struct {
UINT32  CA;             // cached virtual address
UINT32  PA;             // physical address
UINT32  size;            // size,  in MB bytes
} OAL_ADDRESS_TABLE
```

用户为OEM地址映像表填充的数据的最后一个表项必须为全0，以标记地址映像表的结束，但是其中必须至少要有一个非全0的有效的表项存在。每一表项的Virtual Address域的有效数值范围是0x80000000 ~ 0x9FFFFFFF，也就是Windows CE系统虚拟内存的cachable静态映像区域。无论Physical Address域还是Virtual Address域，所有表项所记录的地址空间块之间可以有间隔，但是不能发生重叠。

如果用户在OEM地址映像表中更改了对目标系统上的基本RAM内存的配置，则需要相应地更改与之有关的系统.bib项目档的内容，使这两者相匹配，.bib项目文件中定义系统删内存使用分配时，使用的都是它们在0x80000000 ~ 0x9FFFFFFF范围内的虚拟内存地址。

对于基于x86架构CPU的硬件平台，0x80000000虚拟内存地址必须映像到物理RAM内存的起始地址，而且size域必须以4MB为使用大小。而其余CPU架构类型的硬件平台则没有这样的限制。

## 5.5.3 OEMIoControl函数

OEMGLOBAL结构体的函数指针成员pfnOEMIoctl所服务的oemstub.lib库函数是

OEMIoControl，它所指向的OAL的OEM函数一般也命名为OEMIoControl。OEMIoControl函数负责处理来自操作系统内核的通用型I/O句柄，这个OEM函数实际上就是OAL这个特殊的设备驱动程序的IoControl函数。

OEMIoControl函数的原型定义如下：

```
BOOL OEMIoControl(DWORD code,VOID*pInBuffer,DWORD inSize,VOID*pOutBuffer,DWORD
outsize,DWORD*pOutSize)
```

code参数指明代表特定操作功能的I/O句柄；*pInBuffer和inSize两个参数是OEMIoControl函数的输入参数的缓冲区指针和位组长度；*pOutBuffer和outSize两个参数则是函数输出参数的缓冲区指针和位组长度，pOutSize指针参数所指向的储存单元记录着函数实际输出数据的字节数。OEMIoControl函数的输入参数和输出参数的意义随code参数所记录的句柄的不同而不同，其定义十分灵活。

OEMIoControl函数的原型定义十分灵活，如果不顾忌软件设计上的逻辑合理性只需为它设计不同的句柄，它就可被用来供操作系统内核向OAL的开发者索取任意的数据信息和对目标硬件平台的控制。比如，目前Windows CE操作系统内核可以以IOCTL_HAL_REBOOT为句柄参数读取OEMIoControl函数而重新启动目标嵌入式系统。类似地，OAL的开发者也可以设计一个名为IOCTL_HAL_POWEROFF句柄用来关断目标设备的供电，从而可以代替OEMPowerOff函数。不过Windows CE系统设计并没有这样做，操作系统内核对OAL所需的功能主要都由特定的用户OAL函数，即OAL的OEM函数实现，OEMIoControl函数只用来实现一些其他项的操作功能。

OEMIoControl函数除了供Windows CE操作系统内核的程序代码读取外，还可以为应用程序提供服务。但是Windows Embedded CE 6.0的应用程序都是执行在用户模式的，而OEMIoControl函数作为OAL的OEM函数是在内核模式执行的，所以应用程序不能像系统内核程序代码那样直接读取OEMIoControl函数，而是需要经过一个名为KernelIoControl的系统读取函数的中介转换。

KernelIoControl系统读取函数的原型定义在标头文件%_WINCEROOT%WUBLIC\COMMON\OAK\INC\pkfuncs.h中，其函数原型与OEMIoControl函数完全相同。KernelIoControl函数不做实质性的处理，主要是检查函数参数的有效性，然后依靠读取OEMIoControl函数来实现自身的功能。

由于OEMIoControl函数和它的读取者应用程序都可以是OEM用户开发实现的，所以OEMIoControl函数的实现者可以自定义特殊的句柄用于向应用程序提供特定的功能，只要开发者在它们两者之间，新的句柄所实现的功能就可达成一致。但是如果OEMIoControl函数是为操作系统内核的程序代码提供服务的，则句柄就不是可以随意定义的了，而是由Windows CE系统预定义的。

### 5.5.4 OAL的Cache操作OEM函数

目前的计算机系统，由于CPU的速度越来越快，而其他组件的速度一般都跟不上，即使是速率较低的嵌入式CPU也存在这个问题。而所有这些组件中，对计算机系统执行速度影响最大的是内存，计算机系统的CPU在访问DRAM型内存时通常要插入等待状态，这就拖了整个系统执行速度的后腿。采用SRAM型的内存可以在一定程度上暂时解决这个问题，但是SRAM存在成本高、体积大的缺点，难以大规模应用。为此，目前的计算机系统设计都广泛采用了高速缓冲存储器——Cache，其原理是利用CPU在执行过程中所访问的指令和数据在一定的时间阶段

内通常都局限在一个有限的地址范围，而Cache就是对这个有限的地址范围的内存进行快取的CPU片内SRAM型储存单元。对于设计比较复杂、对系统执行速度要求较高的计算机系统，比如使用x86架构CPU的PC，还使用了二级或者多级Cache的高速缓冲设计。以二级Cache来说，除集成在x86 CPU内的一级Cache（记为L1 Cache）外，还有CPU芯片之外的第二级Cache（L2 Cache），它们都是SRAM型的储存单元，但是后者的储存容量通常比前者高出一个数量级。ARM架构的CPU都是SOC（System On Chip）设计，只有CPU芯片内部的一级Cache，不存在二级Cache。ARM的Cache一般还要分为指令Cache和数据Cache，分别存放CPU执行过程中所访问的程序指令和一般数据。与Cache功能相似的还有TLB（Translation Look-aside Buffer），它是页表专用的Cache，供MMU加速访问页表使用。TLB与Cache没有本质的不同，只不过TLB专门存放页表数据，而普通的Cache存放程序指令和数据。

OEMGLoBAL结构体的各成员中的函数指针型成员pfnCacheRangeFlush实际所指向的是OAL的用于Cache操作的OEM函数。pfnCacheRangeFlush成员所服务的oemstub.lib库函数是OEMCacheRangeFlush，这个OEM函数与pfnCacheRangeFlush成员的类型PFN_CacheRange_Flush有完全相同的函数接口定义，具体如下：

```
Void OEMCacheRangeFlush(LPVOID pAddr,DWORD dwLength,DWORD dwFlags)
```

以上的函数参数中，pAddr是要执行Cache操作的Cache内存的起始地址。dwLength参数则是被执行指定Cache操作的目的Cache内存区域的位组长度。pAddr参数数值NULL和dwLength参数数值0是一对特殊的参数组合，它们代表Cache或者TLB的与dwFlags参数适合的全部内存地址范围，并不是无效的或者不合法的参数数值。dwFlags参数用来供读取者指定要执行的Cache操作的类型，它可能的数值及各自的含义如表5-7所示。

**表5-7 dwFlags参数数值**

| dwFlags参数数值 | 描 述 |
|---|---|
| CACHE_SYNC_WRITEBACK | 将Cache内存中的数据写回内存 |
| CACHE_SYNC_DISCARD | 将Cache内存中的数据写回内存，然后清空Cache内存的特定区域 |
| CACHE_SYNC_INSTRUCTIONS | 丢弃指令Cache内存中的数据 |
| CACHE_SYNC_FLUSH_I_TLB | 清空指令TLB中的储存数据 |
| CACHE_SYNC_FLUSH_D_TLB | 清空数据TLB中的储存数据 |
| CACHE_SYNC_FLUSH_TLB | 指令TLB和数据TLB中的储存数据全部清空 |
| CACHE_SYNC L2_WRITEBACK | 将二级Cache内存中的数据写回内存 |
| CACHE_SYNC_L2_DISCARD | 将二级Cache内存中的数据写回内存，然后清空二级cache内存的特定区域 |
| CACHE_SYNC_ALL | 包括以上的所有操作选项 |

以上的dwFlags参数可能数值的宏定义中的SYNC代表Synchronize，意思是将Cache储单元中的数据写回它当前所映像的内存单元，使这两者同步，汉语翻译为“清理”；而DISCARD意为丢弃，Cache存储单元中的数据不与内存中的同步，翻译成“清除”比较恰当。FLUSH则一般代表先“清理”后“清除”。

# 第6章　接口驱动程序开发

## 6.1　驱动架构分析

### 6.1.1　驱动程序分类

在嵌入式领域，目前还没有像通用计算机领域那样进行了更明细的分工，但是驱动程序也很早就被提出和使用了，Windows Embedded CE 6.0就是使用驱动程序的一个典型例子。Windows Embedded CE 6.0支持多种多样的硬件产品，而且厂商可以为自己的产品开发驱动，让Windows Embedded CE 6.0进行再支持。

按照驱动程序的加载接口不同，驱动程序可以分为本地设备驱动程序和流接口驱动程序。一些类型的设备，如键盘和显示，对操作系统有一个定制的接口，由于它们使用的接口是Windows CE特定的接口，所以这些类型设备的驱动称为本地驱动程序。微软公司为每一种类型的本地设备驱动都定义了定制的接口。虽然每一种类型的设备驱动都有一个定制的接口，但本地设备驱动为特定类型的所有的接口都给出了一组标准的功能，这使Windows CE操作系统以相同的方式对待一种设备类的所有实例，而忽略它们在物理上的差别。

例如，许多Windows CE平台都使用了某种类型的LCD作为显示，可是市场上有各种具有不同操作特征（如分辨率、位深度、内存较差等）的LCD，通过对所有的显示器驱动遵守相同的接口，Windows CE忽略了显示设备本身的物理差别，以相同的方式对待所有的显示设备。

流接口驱动程序也称为可安装的驱动程序，它们是由设备管理器（device.exe）动态加载到用户模式的DLL。流接口驱动程序使用流接口驱动架构，并通过设备管理器或应用程序获得命令。流接口驱动程序包括：PCMCIA驱动、串行端口驱动、ATAFLASH驱动、电池驱动和以太网络驱动等。整个驱动的工作原理如图6-1所示。

### 6.1.2　流接口驱动程序架构

每个流接口驱动程序必须实现一组标准的函数，用来完成标准的文件I/O函数和电源管理函数，这些函数提供给Windows CE操作系统的内核使用。这些函数通常称做流接口驱动程序的DLL接口，如表6-1所示。

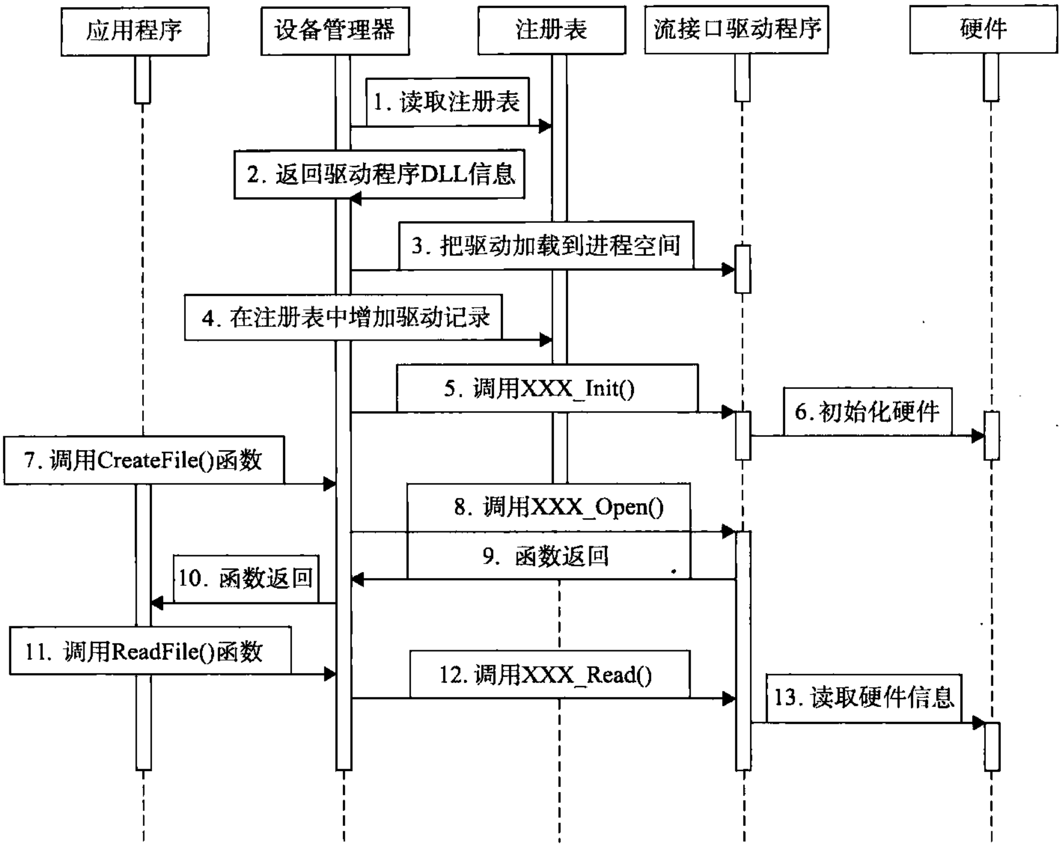



图6-1 流接口驱动程序工作原理顺序图

**表6-1 流接口驱动程序要实现的DLL接口**

| 函数名称 | 描　述 |
|---|---|
| XXX_Init | 当设备管理器初始化一个具体的设备时，调用这个函数 |
| XXX_Deinit | 当设备管理器卸载一个驱动程序时，调用这个函数 |
| XXX_Open | 在打开一个设备驱动程序时，应用程序通过CreateFile()函数调用这个函数 |
| XXX_Close | 在驱动程序关闭时，应用程序通过CloseHandle()函数调用这个函数 |
| XXX_IOControl | 上层的软件通过DeviceIoControl()可以调用这个函数 |
| XXX_PowerUp | 在系统重新启动前，调用这个函数 |
| XXX_PowerDown | 在系统进入暂停状态前，调用这个函数 |
| XXX_Read | 在一个设备驱动程序处于打开状态时，由应用程序通过ReadFile()函数调用这个函数 |
| XXX_Write | 在一个设备驱动程序处于打开状态时，由应用程序通过WriteFile()函数调用这个函数 |
| XXX_Seek | 对设备数据指针进行操作，由应用程序通过SetFilePointer()函数调用这个函数 |

下面分别介绍几个主要的流接口驱动的函数。

1）当加载一个驱动程序时，调用XXX_Init函数，声明如下：

```
DWORD XXX_Init
(
   LPCTSTR pContext,
```

```
   LPCVOID lpvBusContext
)
```

pContext定义一个指向包含有流接口驱动的Active注册表键值路径的字符串，lpvBusContext是一个指向驱动程序特定的结构体，程序开发人员可以通过该参数传递变量。若返回值调用成功，则返回一个驱动程序的句柄。这个函数不能被应用程序调用，当应用程序调用ActiveDeviceEx函数时，设备管理器调用这个函数。

2）当卸载一个驱动程序时，调用XXX_Deinit函数，声明如下：

```
BOOL XXX_Deinit
(
   DWORD hDeviceContext
)
```

hDeviceContext是指向该设备驱动程序的一个句柄，该句柄由XXX_ Init函数返回。若返回值调用成功，则返回TRUE；若失败，则返回FALSE。当一个用户需要卸载一个驱动程序时，设备管理器调用这个函数来卸载这个驱动程序，应用程序不能够直接调用这个函数，可以通过调用DeactivateDevice函数让设备管理器调用该函数。

3）当应用程序通过调用CreateFile来对某个设备进行读/写操作时，会间接调用该函数。声明如下：

```
DWORD XXX_Open
(
   DWORD hDeviceContext,
   DWORD AccessCode,
   DWORD ShareMode
)
```

hDeviceContext是指向该设备的句柄，由XXX_Init函数返回。AccessCode传给驱动程序使用的地址，这个地址跟读/写有关。ShareMode为共享模式，这个参数用于一些专用的设备。例如，某些PC卡的装置读/写是否可以共享。其返回值为驱动程序引用的处理线程。

4）当关闭一个驱动程序的引用实例时，调用XXX_Close函数，声明如下：

```
BOOL XXX_Close
(
   DWORD hOpenContext
)
```

hOpenContext设备驱动的引用实例句柄，由XXX_Open函数返回。若返回值调用成功，则返回TRUE；若失败，则返回FALSE。这个函数用于关闭一个驱动程序的引用线程。使用程序通过CloseHandle()来调用这个函数，当执行完这个函数的时候，hOpenContext将不再有效。

5）当对一个已打开的流接口驱动程序进行读取操作时，可以使用XXX_Read函数，声明如下：

```
DWORD XXX_Read
(
   DWORD hOpenContext,
   LPVOID pBuffer,
   DWORD Count
)
```

hOpenContext是由CreateFile()函数返回的句柄。pBuffer是一个缓冲区地址，用于从驱动读数据。Count需要读取缓冲区的长度。返回值是实际读取字节的长度。

6）当对一个已打开的流接口驱动程序进行写入操作时，可以使用XXX_Write函数，声明如下：

```
DWORD XXX_Write
(
   DWORD hOpenContext,
   LPVOID pBuffer,
   DWORD Count
)
```

hOpenContext是由CreateFile()函数返回的句柄。PBuffer是一个缓冲区地址，用于从驱动写入数据。Count是需要写入缓冲区的长度。返回值是实际写入字节的长度。当一个流接口驱动程序打开以后，可以使用WriteFile()函数进行写入操作。

7）当应用程序调用SetFilePointer函数时，操作系统会调用XXX_Seek函数。声明如下：

```
DWORD XXX_Seek
(
   DWORD hOpenContext,
   long Amount,
   WORD Type
)
```

hOpenContext是由XXX_Open函数返回Amount定义要移动的长度，正数表示向后移动，负数表示向前移动。Type定义指标的起始点，它可以是FILE_BEGIN、FILE_CUTTENT或FILE_END。

8）当应用程序向设备发送一个命令时，可以使用XXX_IOControl函数，声明如下：

```
BOOL XXX_IOControl
(
   DWORD hOpenContext,
   DWORD dwCode,
   PBYTE pBufIn,
   DWORD dwLenIn,
   PBYTE pBufOut,
   DWORD dwLenOut,
   PDWORD pdwActualOut
)
```

hOpenContext是由CreateFile()函数返回的句柄。DwCode是用于描述这次IOControl操作的语义，一般都由用户自定义。pBufIn缓冲区指标指向需要传送给驱动程序使用的数据。DwLenIn是要传送给驱动程序使用数据的长度。PBufOut是缓冲区指针指向驱动程序应用程序使用的数据。DwLenOut是要传送给应用程序使用数据的长度。PdwActualOut是DWORD型指针用于返回实际处理数据的长度。返回值，调用成功返回TRUE，调用失败返回FALSE。这个函数通常用于向设备发送一个命令。应用程序使用DeviceIOControl函数来通知操作系统调用这个函数。通过参数dwCode来通知驱动程序要执行的操作。这个函数扩充了流接口驱动程序的功能。

9）在恢复对设备的供电时，会调用XXX_PowerUp函数，声明如下：

```
void XXX_PowerUp
```

```
(
   DWORD hDeviceContext
)
```

hDeviceContext是由XXX_Init建立时生成的设备句柄。PowerUp函数通常都必须要硬件的支持才能有效，也就是说，相关的硬件必须支持PowerUp模式。

10）当系统处于暂停状态时，会调用XXX_PowerDown函数，声明如下：

```
void XXX_PowerDown
(
   DWORD hDeviceContext
)
```

hDeviceContext是由XXX_Init新建时生成的设备句柄。PowerDown函数通常都必须要硬件的支持才能有效，也就是说，相关的硬件必须支持PowerDown模式。

### 6.1.3 添加一个流驱动到BSP中

以对实验平台上的LED灯的添加控制驱动为例，介绍一个流驱动的添加过程。由于系统已经有一个名为LED的LED灯驱动，现在添加的LED灯的驱动功能，将与之前的驱动功能一致，在这里只是修改一下该驱动的名称，如LET。

首先，在Windows Embedded CE 6.0安装目录PLATFORM\DMA6410XP\src\Drivers下，有很多文件夹，这些文件夹都跟驱动有关。找到名称为Led的文件夹，进入该文件夹下可以看到有LED.c、makefile、LED.def、sources四个文件，这些文件都是与驱动有关的文件。

1）在Drivers资料夹下新建一个名为LET的文件夹，将Led文件夹下的LED.c、makefile、LED.def、sources四个文件复制到LET文件夹中。然后，将LED.def重新命名为LET.def，LED.c重新命名为LET.c。

2）用记事本打开文档sources，修改其中的TARGETNAME=LED为TARGETNAME=LET；修改DEFFILE =LED.def为DEFFILE=LET.def；修改SOURCES=\LED.c为SOURCES=\LET.c。

3）打开LET.def，修改LIBRARY NAV_LED为LIBRARY LET，依此类推，修改LED_Init、LED_Deinit等的LED为LET，即LET_Init、LET_Deinit……

4）打开在Drivers文件下的dirs文件可以看到“DIRS=ceddk \ keybd \ PowerButton \ pccard \ serial \ usb \ Battdrvr \ Backlight…”这样的一段程序代码，在最后添加上LET，记得留意其中的格式。

5）在DMA6410XP\Files文件夹下打开platform.bib文件找到“NAV_LED.dll $（_FLATRELEASEDIR）\NAV_LED.dll NK SHK”这样的一行程序代码，这是LED驱动的程序代码，复制这一行程序代码，然后将复制程序代码中的NAV_LED.dll修改为LET.dll。

6）继续在该文件夹下打开platfor.reg文件，找到下面的这一段程序代码：

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\led]
        "Dll"="NAV_LED.dll"
        "Prefix"="LED"
        "Order"=dword:0
        "Index"=dword:1
```

同样将这段程序代码复制一份，将程序代码修改为如下：

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\LET]
        "Dll"="LET.dll"
        "Prefix"="LET"
        "Order"=dword:0
        "Index"=dword:1
```

7）重新打开系统项目，打开类似如图6-2所示的文件目录。在原来LED的驱动的文件夹下应该多出了一个LET文件夹，打开其中的LET.c，依次修改这里的LED_Init、LED_Open等10个函数名称为LET_Init、LET_Open等，和第3步修改LET.def类似。找到程序代码“#define LED_DBG_MSG 0”，并修改为“#define LED_DBG_MSG 1”，如果这里的0已经是1了，那么不用修改它，到LED文件夹下，打开LED.c，修改其中的这句程序代码中的1为0。这个宏定义是控制打印的开关，关掉原来LED驱动中的打印，打开新LET驱动的打印，在驱动执行的时候就可以看到相关的打印信息。

8）重新编译当前系统。重新下载NK执行系统。这时注意系统启动时的打印信息，可以看到类似如图6-3所示的打印信息。

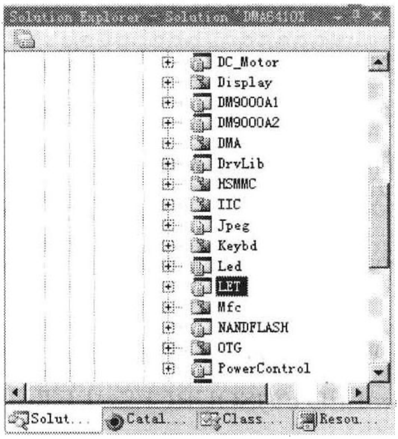



图6-2 驱动程序目录

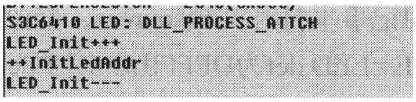



图6-3 编译当前系统提示信息

此时，驱动已经成功加入到了系统中。为了进一步验证驱动程序，可以制作一个应用程序，或把LED驱动程序的原始程序代码打开，修改“m_hled = CreateFile（TEXT（"LED1:"），GENERIC_ READ | GENERIC_WRITE，0，NULL，OPEN_EXISTING，0，0）;”中的LED1为LET1，然后重新编译驱动程序，如果该驱动程序也能正常地控制LED灯，那么说明驱动制作成功。

## 6.2 LED接口及驱动程序

在DMA-6410XP实验平台上，配置有两个LED发光二极管，分别由输入/输出接口GPN14控制LED1，GPN15控制LED2。采用Visual Studio 2005编写LED驱动程序，在Windows Embedded CE 6.0下实现对两个LED发光二极管的控制，并将结果显示在LCD屏幕上。

### 6.2.1 LED电路设计

LED即发光二极管，是嵌入式系统中常用的输出设备。单个LED通常做报警显示、故障显示或信号显示等。LED是一种半导体组件，当电流通过它的时候，可以产生可见光。LED的发光强度与通过的电流强度成正比，LED发光的颜色可以是红、黄、绿或者蓝色。LED电路图如图6-4所示。

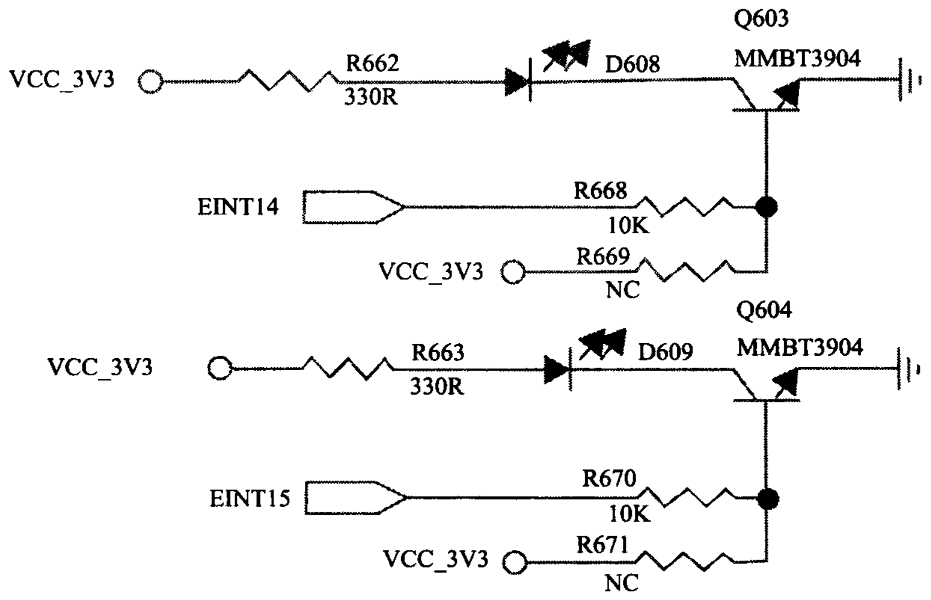



图6-4 LED电路图

### 6.2.2 S3C6410 GPIO专用寄存器

由GPN14和GPN15控制两个LED指示灯。端口N控制寄存器包括3个控制寄存器，如表6-2所示，分别是GPNCON、GPNDAT、GPNPUD。

**表6-2 端口N控制寄存器的3个控制寄存器**

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| GPNCON | 0x7F008830 | 读/写 | 端口N 配置寄存器 | 0x00 |
| GPNDAT | 0x7F008834 | 读/写 | 端口N数据寄存器 | 未定义 |
| GPNPUD | 0x7F008838 | 读/写 | 端口N上拉/下拉寄存器 | 0x55555555 |

1）端口N控制寄存器如表6-3所示。

**表6-3 端口N控制寄存器**

| GPNCON | 位 | 描述 | 初始状态 |
|---|---|---|---|
| GPN14 | [29:28] | 00=输入 01=输出<br>10=外部中断[14] 11=保留 | 00 |
| GPN15 | [31:30] | 00=输入 01=输出<br>10=外部中断[15] 11=保留 | 00 |

2）端口N数据寄存器如表6-4所示。

3）端口N上拉/下拉寄存器如表6-5所示。

表6-4 端口N数据寄存器

| GPNDAT | 位 | 描 述 |
| --- | --- | --- |
| GPN[15:0] | [15:0] | 当端口作为输入端口时，相应的位引脚状态。当端口作为输出引脚时，引脚状态等同于相应的位。当端口作为功能引脚时，读取未被定义的值 |

表6-5 端口N上拉/下拉寄存器

| GPNPUD | 位 | 描 述 |
| --- | --- | --- |
| GPN[n] | [2n+1:2n]<br>n=0～15 | 00=禁止上拉/下拉<br>01=下拉使能<br>10=上拉使能<br>11=保留 |

## 6.2.3 LED驱动程序分析

通过对电路图和有关寄存器的分析，两种状态分别由GPN14和GPN15来控制。首先设置GPNCON14和GPNCON15为“01”（Output），然后通过设置GPNDAT的对应位来控制D608和D609的亮与灭，若设置为“1”，则灯亮；若设置为“0”，则灯灭。

LED驱动程序是根据流驱动接口实现的，所用到函数如表6-6所示。

表6-6 流接口驱动函数

| 函 数 | 描 述 |
| --- | --- |
| LED_Init | 驱动初始化（loaded） |
| LED_Open | 应用程序调用CreateFile |
| LED_Close | 应用程序调用CloseHandle |
| LED_IOControl | 应用程序调用DeviceIoControl |
| LED_Read | 应用程序调用ReadFile |
| LED_Write | 应用程序调用WriteFile |
| LED_Seek | 应用程序调用SetFilePointer |
| LED_PowerUp | 操作系统从暂停状态中恢复 |
| LED_PowerDown | 操作系统进入暂停状态 |

LED驱动的操作流程方块图如图6-5所示。

流驱动入口函数DllMain()，主要接收进程和线程附加消息。接下来，LED_Init包含了InitLedAddr（完成物理地址到虚拟地址的映射）和GPIOInit（初始化寄存器，设置GPN14和GPN15为输出模式）两个函数。LED_IOControl由上层应用程序DeviceIOControl函数调用，接受上层应用程序传递过来的控制字和参数，控制对应的LED亮灭，包含了ProcLed（对具体的某个灯进行操作）函数。

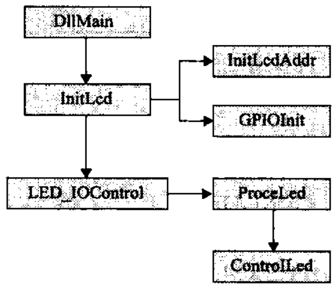


图6-5 LED驱动操作流程图

## 6.2.4　LED底层驱动程序设计

LED底层驱动程序设计代码如下：

```
#include <windows.h>
#include <s3c6410.h>
#include <ceddk.h>
#include "bsp_cfg.h"

#define LED_DBG_MSG  0
#define LED_ERR_MSG       0

typedef enum _LED_IO_CONTROL_VALUE {
    LED_VERSION,
    IO_LED_ON     = 0x10,
    IO_LED_OFF    = 0x11,
} ENUM_LED_IO_CONTROL, *PENUM_LED_IO_CONTROL;

volatile S3C6410_GPIO_REG *v_pIOP_LED_regs = NULL;

#define     GPN14          28
#define     GPN15          30

#define     LED1           GPN14  //D608
#define     LED2           GPN15  //D609

// LED控制位

#define     LED1_BIT   14
#define     LED2_BIT   15

#define     Led_On(x)      (v_pIOP_LED_regs->GPNDAT |= (0x1 << (x)))
#define     Led_Off(x)(v_pIOP_LED_regs->GPNDAT &= ~(0x1 << (x)))

void  InitLedAddr()
{
   DWORD dwIOBase;
   DWORD dwIOSize;
   PHYSICAL_ADDRESS  IOPhyAdr= { S3C6410_BASE_REG_PA_GPIO,0 };
   RETAILMSG(LED_DBG_MSG, (TEXT("++InitLedAddr\r\n")));
   dwIOSize = sizeof(S3C6410_GPIO_REG);
   if(NULL == v_pIOP_LED_regs)
   {
          v_pIOP_LED_regs = (volatile S3C6410_GPIO_REG *)MmMapIoSpace(IOPhyAdr,
(ULONG)dwIOSize,FALSE);
          if(NULL == v_pIOP_LED_regs)
          {
                 RETAILMSG(LED_ERR_MSG, (TEXT("LED init address failed\r\n")));
          }
   }
```

```
    }

    void GPIOInit()
    {

        if(v_pIOP_LED_regs)
        {
                //GPN14,GPN15
                v_pIOP_LED_regs->GPNPUD &= ~((0x3<<LED1)|(0x3<<LED2));/*设置GPN14、GPN15上拉*/
                v_pIOP_LED_regs->GPNCON &= ~((0x3<<LED1)|(0x3<<LED2));
                v_pIOP_LED_regs->GPNCON |= (0x1<<LED1)|(0x1<<LED2);//设置GPN14、GPN15为输出

                v_pIOP_LED_regs->GPNDAT |= ~((0x1<<LED1_BIT)|(0x1<<LED2_BIT));/*点亮led1,关闭led2*/
        }
    }

    void ControlLed(UINT8 u8Index,BOOL bHigh)
    {
        if(bHigh)
                Led_On(u8Index);
        else
                Led_Off(u8Index);
    }

    BOOL ProcLed(UINT8 u8Index,BOOL bHigh)
    {
        RETAILMSG(LED_DBG_MSG, (TEXT("ProcLed+++\r\n")));
        if(u8Index>2)
        {
                RETAILMSG(LED_ERR_MSG, (TEXT("input error,exit function")));
                return FALSE;
        }

        switch(u8Index)
        {
                case  1:
                        ControlLed(LED1_BIT,bHigh);
                        RETAILMSG(LED_DBG_MSG, (TEXT("******v_pIOP_LED_regs->GPNCON=%x\
r\n"),v_pIOP_LED_regs->GPNCON));
                        RETAILMSG(LED_DBG_MSG, (TEXT("******v_pIOP_LED_regs->GPNDAT=%x\
r\n"),v_pIOP_LED_regs->GPNDAT));
                        break;
                case  2:
                        ControlLed(LED2_BIT,bHigh);
                        RETAILMSG(LED_DBG_MSG, (TEXT("******v_pIOP_LED_regs->GPNCON=%x\
r\n"),v_pIOP_LED_regs->GPNCON));
                        RETAILMSG(LED_DBG_MSG, (TEXT("******v_pIOP_LED_regs->GPNDAT=%x\
r\n"),v_pIOP_LED_regs->GPNDAT));
                break;
```

```
            default:
                    break;
    }
    RETAILMSG(LED_DBG_MSG, (TEXT("Operate LED: %d\r\n"),u8Index));
    RETAILMSG(LED_DBG_MSG, (TEXT("ProcLed---\r\n")));
    return TRUE;
}
BOOL WINAPI DllMain(HANDLE hinstDll,ULONG Reason,LPVOID Reserved)
{
    switch(Reason)
    {
            case DLL_PROCESS_ATTACH:
                    RETAILMSG(LED_DBG_MSG,(TEXT("S3C6410 LED: DLL_PROCESS_ATTCH\r\n")));
                    break;

            case DLL_PROCESS_DETACH:
                    RETAILMSG(LED_DBG_MSG,(TEXT("S3C6410 LED: DLL_PROCESS_DETACH\r\n")));
                    break;

            default:
                    break;
    }

    return TRUE;
}

BOOL LED_IOControl(DWORD dwOpenContext,DWORD dwIoControlCode,LPBYTE lpInBuf,
                   DWORD nInBufSize,LPBYTE lpOutBuf,DWORD nOutBufSize,
                   LPDWORD lpBytesReturned)
{
    RETAILMSG(LED_DBG_MSG,(TEXT("LED_IOControl+++\r\n")));
    RETAILMSG(LED_DBG_MSG,(TEXT("dwIoControlCode = %d.\r\n"),dwIoControlCode));
    switch(dwIoControlCode)
    {
            case IO_LED_ON:
                    if(NULL != lpInBuf)
                    {
                            ProcLed(*lpInBuf,TRUE);
                            RETAILMSG(LED_DBG_MSG,(TEXT("LED_IOControl: IO_LED_ON\r\n")));
                    }
                    else
                            RETAILMSG(LED_ERR_MSG, (TEXT("The lpInBuf is NULL\r\n")));
                    break;

            case IO_LED_OFF:
                    if(NULL != lpInBuf)
                    {
                            ProcLed(*lpInBuf,FALSE);
                            RETAILMSG(LED_DBG_MSG,(TEXT("LED_IOControl: IO_LED_OFF\r\n")));
```

```
                }
                else
                        RETAILMSG(LED_ERR_MSG, (TEXT("The lpInBuf is NULL\r\n")));
                break;

        default:
                RETAILMSG(LED_ERR_MSG,(TEXT("LED_IOControl: default\r\n")));
                break;
    }
    RETAILMSG(LED_DBG_MSG,(TEXT("LED_IOControl---\r\n")));
    return TRUE;
}
```

## 6.2.5 LED驱动程序设计

### 1. 软件框架设计

LED的驱动程序实现的功能是控制实验平台上LED灯的亮与灭。这里有两个LED灯，可以设计使用4个按钮来控制它们的亮与灭，两个用于分别点亮两个灯，两个用于分别熄灭两个灯。首先新建一个基于对话框的MFC项目。

### 2. 建立LED工程

首先打开计算机桌面上的Visual Studio 2005，选择“File”→“New”→“Project”后弹出如图6-6所示对话框，在“Project types：”中选择Visual C++下的“Smart Device”，在“Templates:”中选择“MFC Smart Device Application”。在“Name：”后的文本框中输入“LED”。最后单击“OK”按钮，开始建立工程，如图6-6所示。

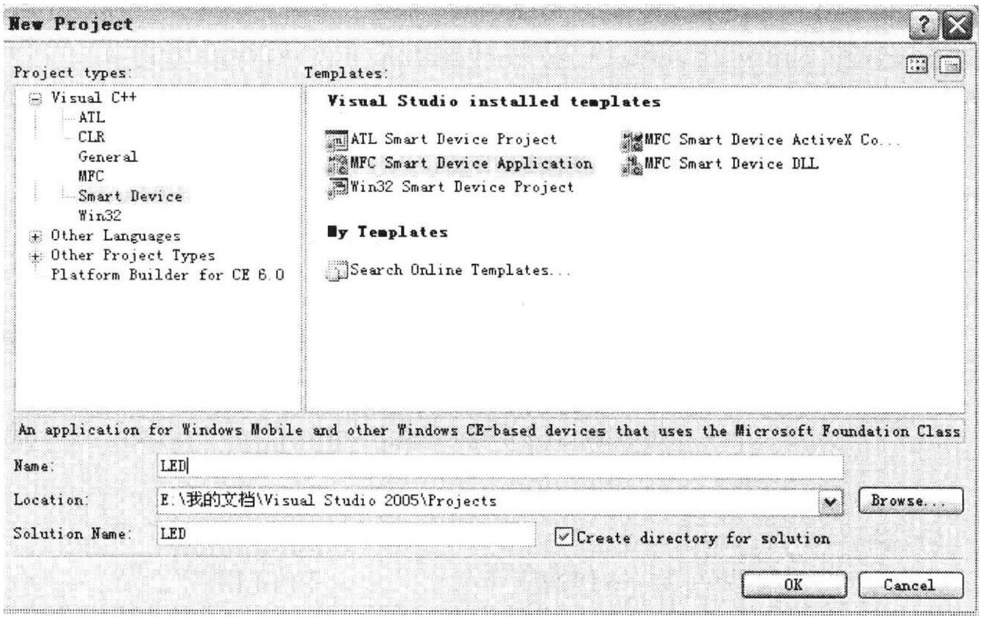



图6-6 建立LED工程

在“Overview”界面中选择“Next”，如图6-7所示。

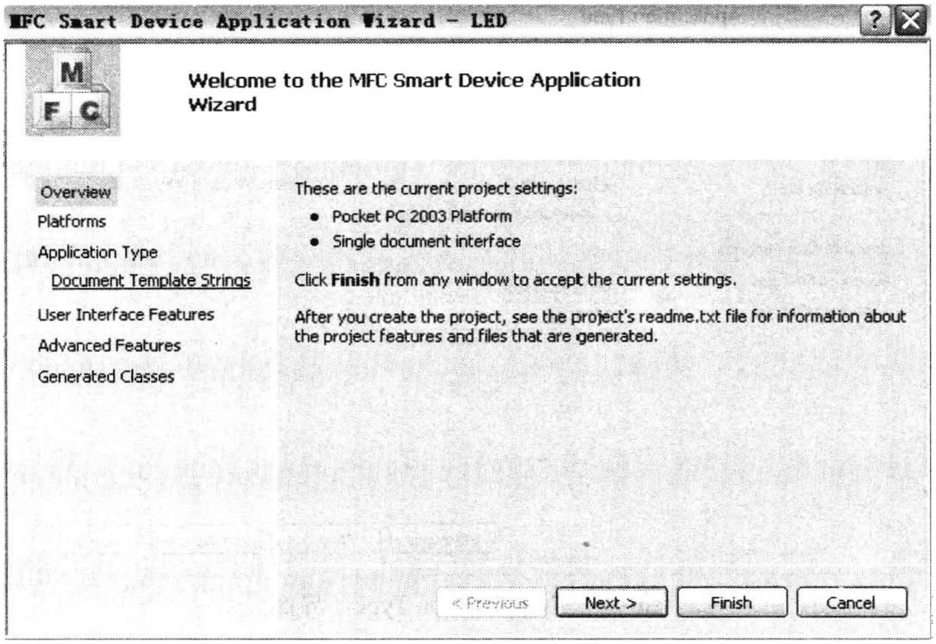



图6-7　“Overview”界面

在“Platforms”界面中选择“DMA6410”的SDK加入到“Selected SDKs”中，并移除“Selected SDKs”中的“Pocket PC 2003”，如图6-8所示。完成后单击“Next”按钮进入下一步。

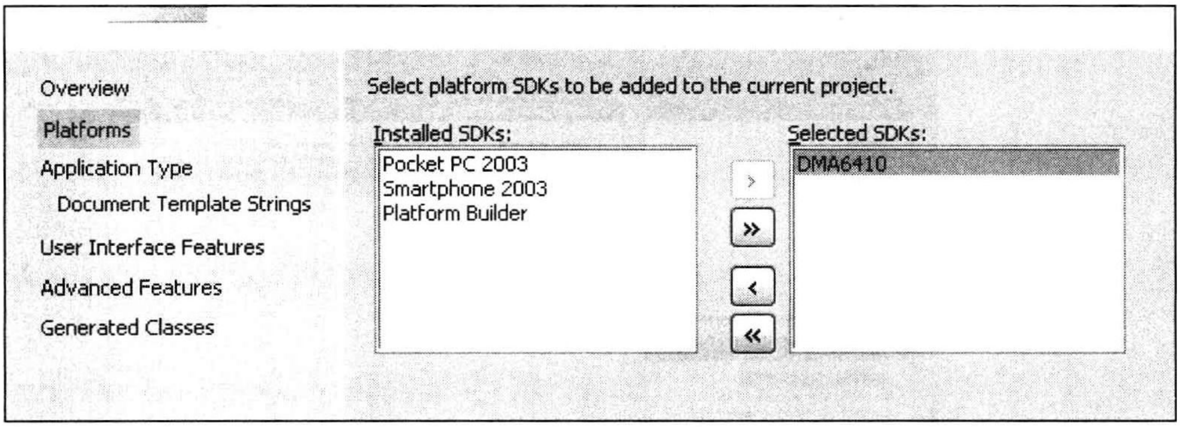



图6-8　修改SDK

在“Application Type”界面中，在“Applicatoin type”栏中选择“Dialog based”，在“Use of MFC”栏中选择“Use MFC in a static library”，在“Resource Language”选择“中文（中国）”，如图6-9所示。完成后单击“Next”按钮。

新建一个名为Led的智能设备上的MFC项目。项目建立好后有如图6-10所示的界面。

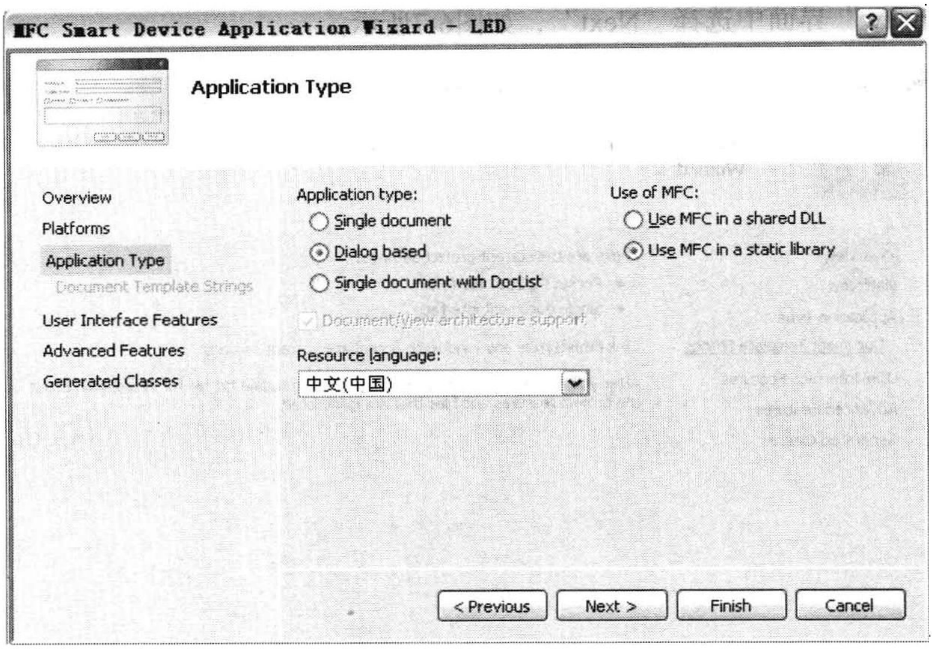



图6-9 “Application Type”界面

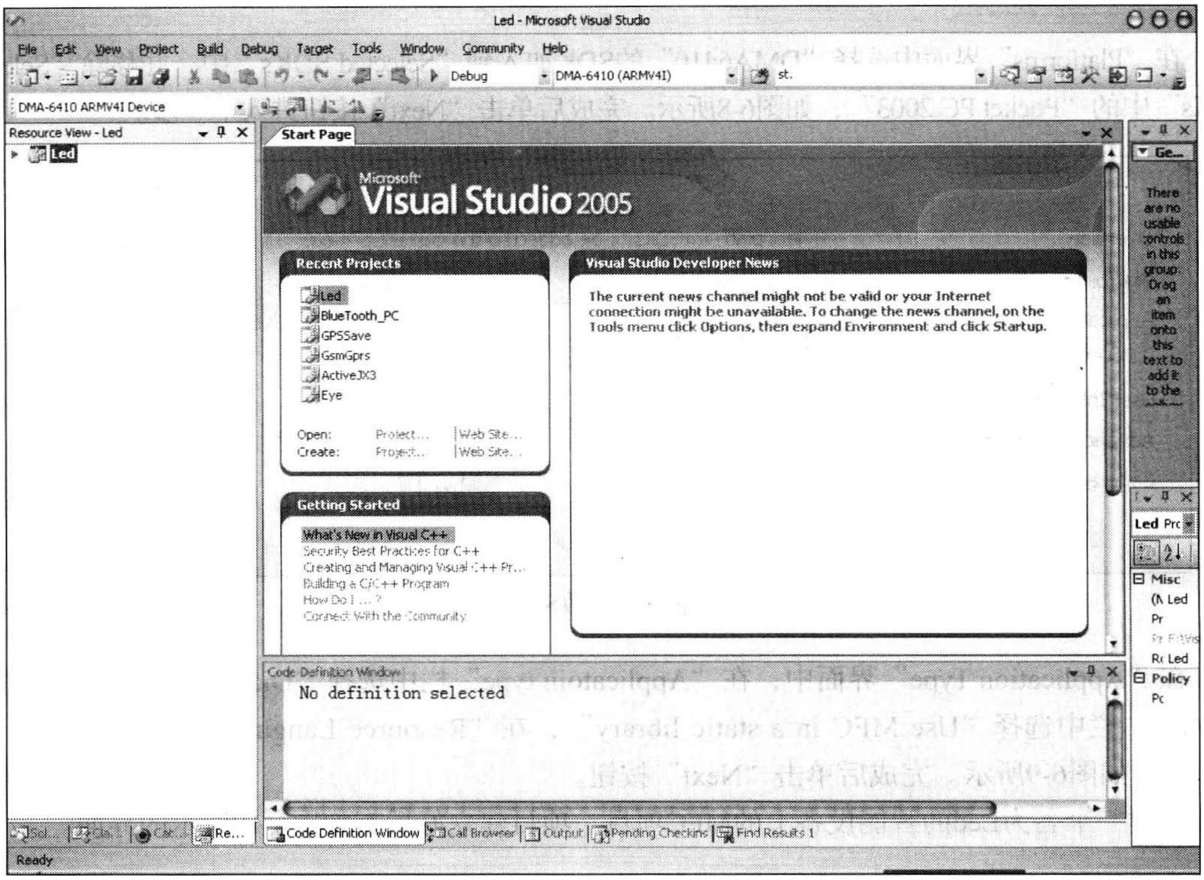



图6-10 新建Led的MFC项目

### 3. 设计界面

依次选择菜单项“View”→“Resource View”，如图6-11所示。

然后会出现一个资源视图窗口，在该窗口中，依次展开“Led”→“Led.rc”→“Dialog”，然后单击两下“IDD_LED_DIALOG”，出现如图6-12所示的界面。

在中间部分有一个对话框造型的界面，这就是程序执行后的界面，只需要在上面添加两个按钮即可。

首先，用鼠标选择“TODO：在此放置对话框控件。”这行文字，这个是系统自动添加的一个STATIC控件，并不需要它，按“Delete”键删除它。

接下来，选择菜单“View”→“Toolbox”项，将出现一个“控件”窗口，用鼠标选择其中的“Button”控件，如图6-13所示。

然后移动鼠标到对话框的窗体上进行选择，将出现一个按钮，可以移动和更改该按钮的位置和大小，如图6-14所示。

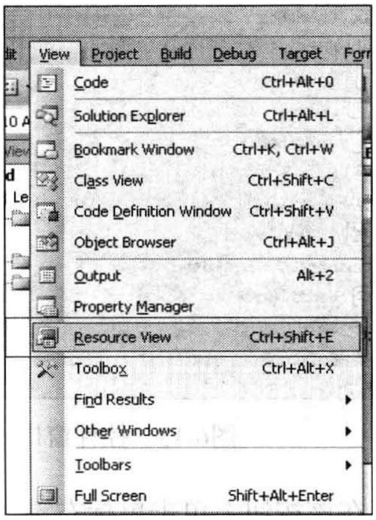



图6-11 选择“Resource View”菜单项

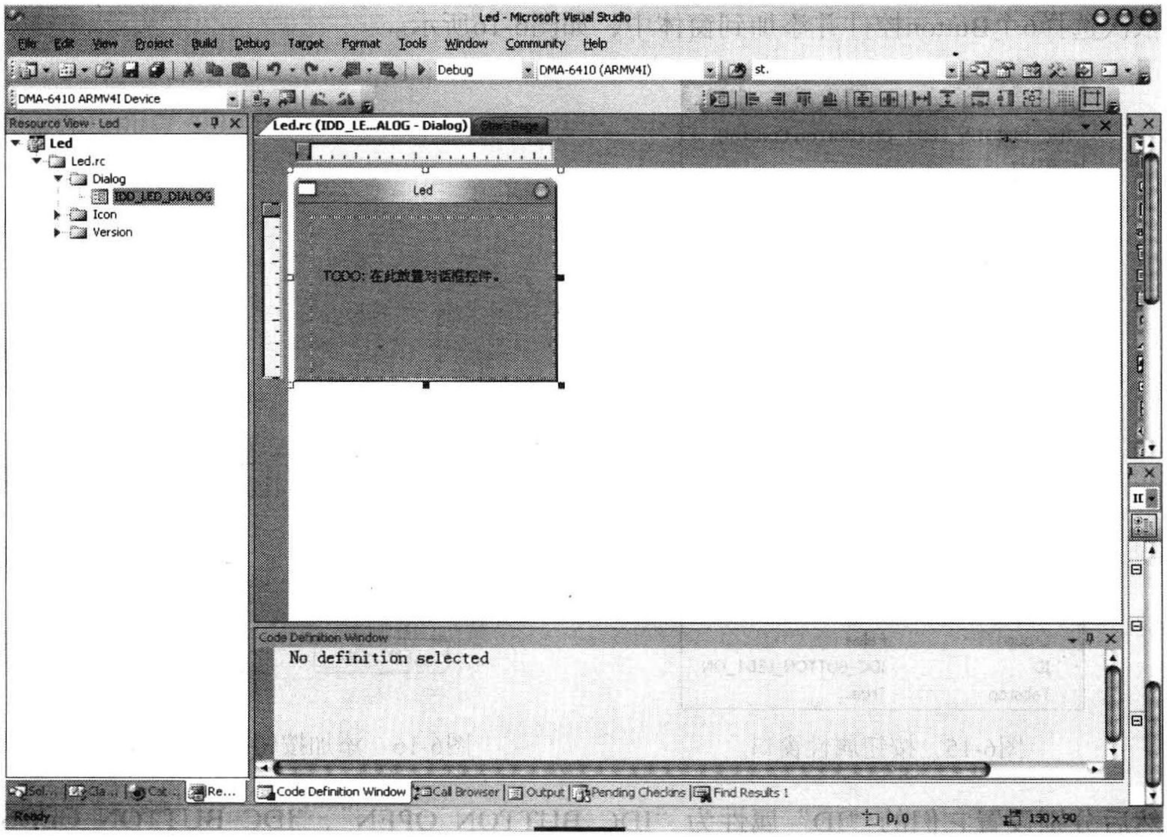



图6-12 对话框界面

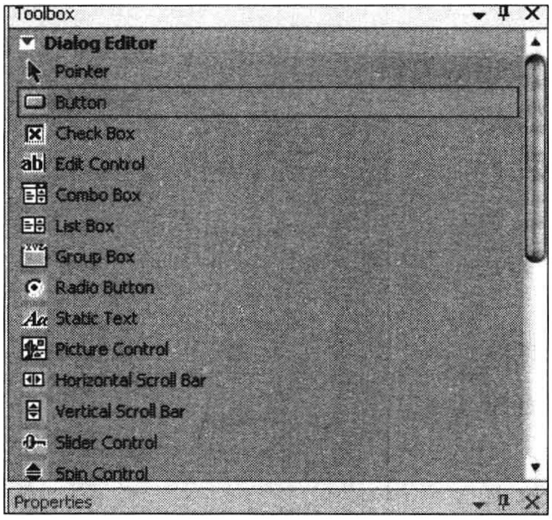



图6-13 控件窗口

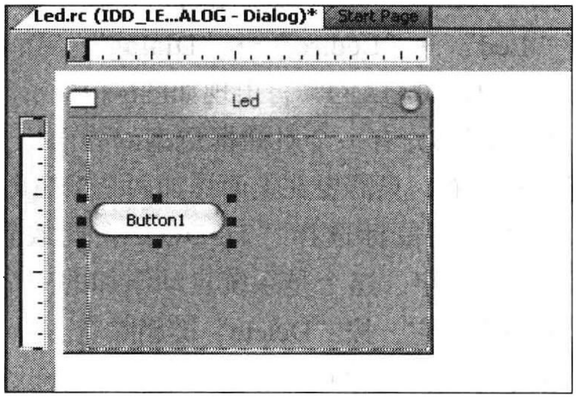



图6-14 按钮控件

在该按钮上单击鼠标右键，在弹出的快捷菜单中选择“Properties”项，将出现该按钮的属性窗口，在这里可以设置该按钮的一些属性。例如，如果希望把这个按钮作为一个点亮LED灯的按钮，那么需要设置以下两个属性，即设置“ID”属性为“IDC_BUTTON_LED1_ON”，设置“Caption”属性为“LED1_ON”，如图6-15所示。

依次选择6个Button控件并添加到窗体中，如图6-16所示。

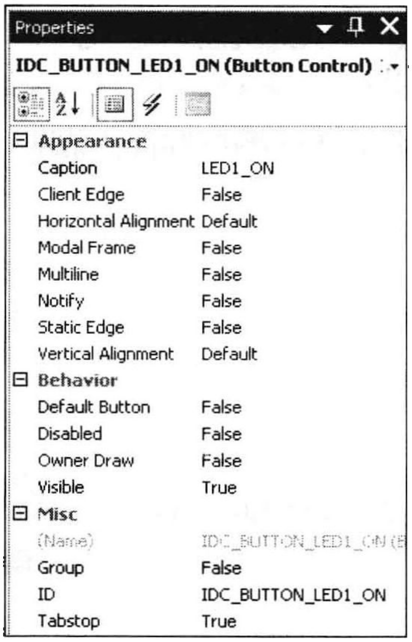



图6-15 按钮属性窗口

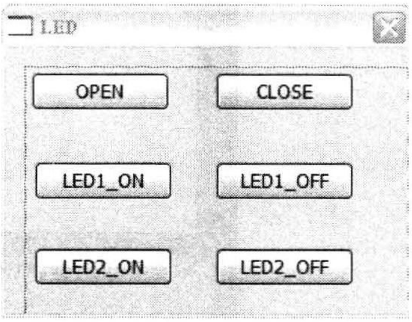



图6-16 添加按钮控件到窗体中

然后分别设置它们的“ID”属性为“IDC_BUTTON_OPEN”、“IDC_BUTTON_CLOSE”、“IDC_BUTTON_LED1_ON”、“IDC_BUTTON_LED1_OFF”、“IDC_BUTTON_LED2_ON”和

“IDC_ BUTTON_LED2_OFF”。

接着分别设置它们的“Caption”属性为“Open”、“Close”、“LED1_ON”、“LED1_OFF”、“LED2_ON”和“LED2_OFF”，如表6-7所示。

表6-7　Caption和ID属性

| 控件 | Caption | ID |
|---|---|---|
| Button1 | Open | IDC_BUTTON_OPEN |
| Button2 | Close | IDC_BUTTON_CLOSE |
| Button3 | LED1_ON | IDC_BUTTON_LED1_ON |
| Button4 | LED1_OFF | IDC_BUTTON_LED1_OFF |
| Button5 | LED2_ON | IDC_BUTTON_LED2_ON |
| Button6 | LED2_OFF | IDC_BUTTON_LED2_OFF |

### 4. 编译软件

先来编译一下。在Visual Studio 2005上方的工具栏中间有编译方式和SDK选择两个下拉列表框，选择编译方式为“Release”。SDK则选择建立项目时添加的那个SDK，如图6-17所示。

配置好这两个选项后，依次选择菜单项“Build”→“Rebuild Led”，开始编译项目，如图6-18所示。

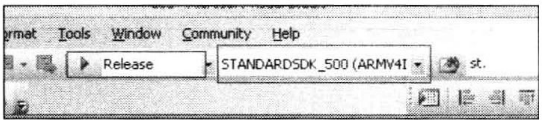



图6-17　选择编译方式

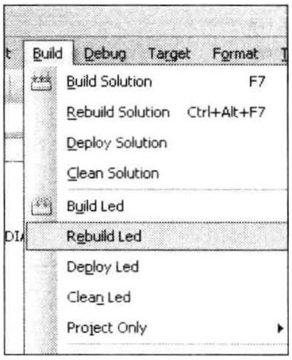



图6-18　运行编译项目

编译完成后，在“Output”窗口中有如下的输出信息。

```
1>Build log was saved at "file://f:\Visual Studio 2005\Projects\6410XP AP\Led\I
1>Led - 0 error(s), 0 warning(s)
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

其中，第一行指示了编译的日志文件存放的路径，需留意这个路径，在这个路径下同样可以找到一个名为“Led.exe”的文件。该文件就是编写程序的可执行文件。通过SD卡复制这个程序到实验平台上，可以放在任何目录下，单击两下该文件，则有如图6-19所示的软件界面。

下面将介绍如何为这6个按钮添加程序代码来控制LED灯的亮灭。

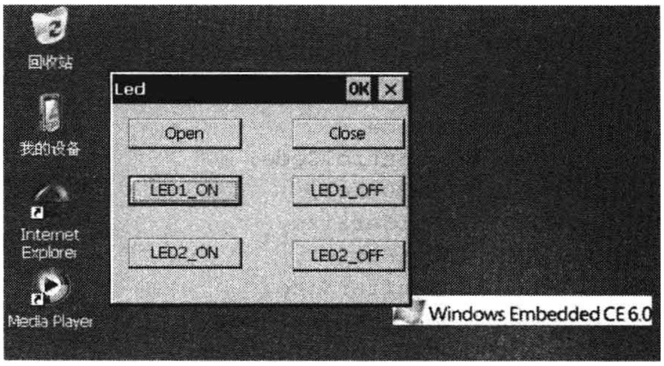



图6-19　LED软件界面

#### 1. 添加代码

在“View”→“Solution Explorer”中找到LEDDlg.cpp文件并双击，打开LEDDlg.cpp文件，在文件中添加如下驱动程序代码，该段代码用来调用驱动中的LED控制功能，如图6-20所示。

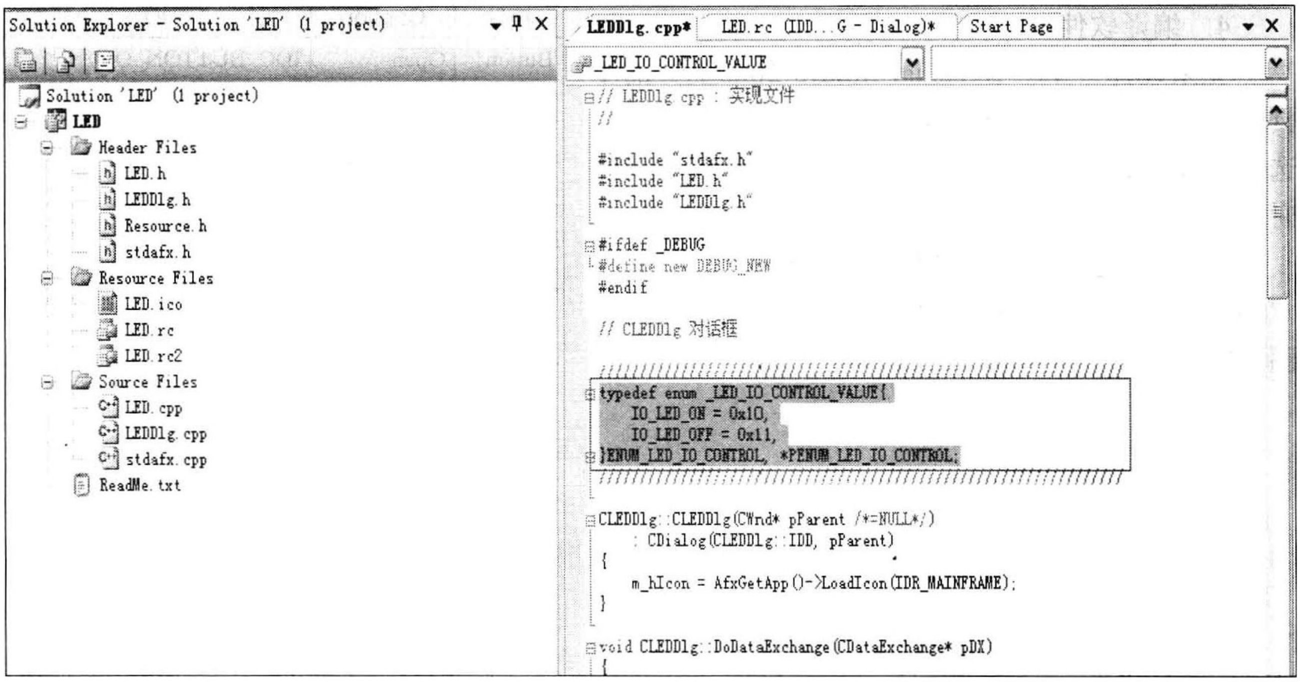


图6-20 驱动代码

在程序中，定义LED驱动：

```
typedef enum _LED_IO_CONTROL_VALUE {
    IO_LED_ON     = 0x10,
    IO_LED_OFF    = 0x11,
} ENUM_LED_IO_CONTROL, *PENUM_LED_IO_CONTROL;
```

这里的“IO_LED_ON”为“亮”，“IO_LED_OFF”为“不亮”。然后，在驱动的LED_IOControl函数中有“case IO_LED_ON:”和“case IO_LED_OFF:”分支语句，根据需求去执行相应的开关灯的动作。

在应用中，DeviceIoControl函数的声明如下：

```
BOOL DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

需要注意此函数的前3个参数：hDevice、dwIoControlCode和lpInBuffer。

- hDevice是一个设备句柄。
- dwIoControlCode所带的参数是IO_LED_ON或IO_LED_OFF，告诉驱动开灯或关灯。
- lpInBuffer是一个指标参数，在这里可以传递个数。

CreateFile函数的声明如下：

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

需要注意此函数的第一个参数lpFileName。

lpFileName表示了驱动的名称，针对本例，则是“LED1:”。那么，为什么是“LED1:”呢？先来看看添加LED驱动的注册表中的几行代码：

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\led]
   "Dll"="LED.dll"
   "Prefix"="LED"
   "Order"=dword:0
   "Index"=dword:1
```

其中的Dll表示LED驱动加载的DLL的名称；Prefix是指LED驱动的名称；Order表示驱动加载的顺序；Index表示驱动名的一个索引，所以这里要使用“LED1：”，至于最后的一个“：”，这只是一种语法格式。

上层应用是如何调用底层驱动的呢？

首先，需要使用CreateFile函数来获取该驱动的设备句柄（HANDLE），在该函数的第一个参数中需要指定驱动的名称（LED1：）。

接下来，获得设备句柄（HANDLE）之后，使用DeviceIoControl传递参数给驱动，让驱动去做某件事情。DeviceIoControl函数的第一个参数要使用刚才获取的设备句柄；第二个参数是一个句柄，IO_LED_ON表示点亮LED灯，IO_LED_OFF表示关闭LED灯；第三个参数是通知驱动点亮或关闭哪个LED灯。

对于上层应用控制LED灯，只需要两个步骤。现在已经有了写DeviceIoControl函数的地方——LED1_ON按钮，但是还没有获取到设备句柄，那么就再添加一个按钮来获取设备句柄，把这个按钮称为Open，打开驱动。

### 2. 定义设备句柄变量

选择菜单项“View”→“Solution Explorer”，将出现一个“Solution Explorer”窗口，在该窗口中展开“LED”→“Header Files”，然后单击两下“LEDDlg.h”文件，这是“LEDDlg.cpp”的头文件。在这个文件中添加一个HANDLE的变量，添加在OnBnClickedButtonLed1On函数的声明中，如图6-21所示。

```
public:
                    HANDLE          m_hled;         //设备句柄
```

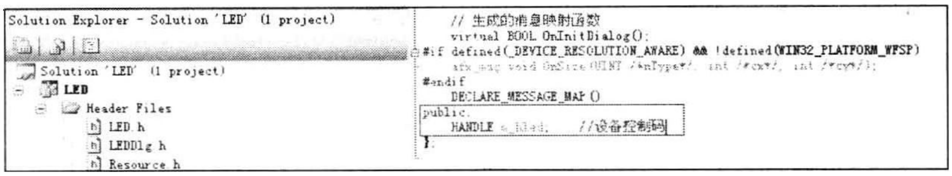



图6-21 添加HANDLE变量

### 3. 给按钮添加点选事件

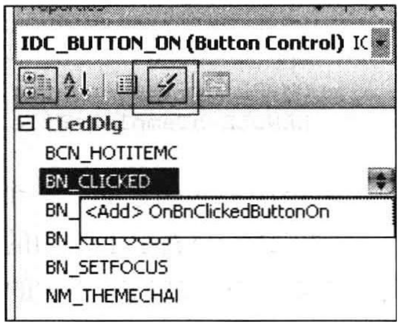



图6-22 给按钮添加点选事件

首先找到这个函数位置。以“LED1_ON”按钮为例，在该按钮上单击鼠标右键，然后选择“Properties”项，在出现的属性窗口中选择[⚡]图标按钮，切换到按钮的事件列表中，在“BN_CLICKED”中选择“<Add>OnBnClickedButtonOn”，添加该按钮的选择事件处理函数，如图6-22所示。

当选择了“<Add>OnBnClickedButtonOn”后，Visual Studio 2005会自动添加一个OnBnClickedButtonLed1On函数，并在中间原来显示窗体的地方自动显示出该段程序代码。当选择一次这个按钮的时候，这个OnBnClickedButtonLed1On函数中的程序代码就会被“从上至下”的执行一次。接下来就要在这个函数中添加程序代码来控制LED1灯亮。

### 4. 编写“Open”按钮程序代码

在“Resource View”窗口中依次展开“LED”→“LED.rc”→“Dialog”，然后双击IDD_LED_DIALOG，在窗体中双击Open按钮，添加按钮Click事件，进入代码模式。在OnBnClickedButtonOpen()函数中添加以下代码：

```
void CLEDDlg::OnBnClickedButtonOpen()
{
    m_hled = CreateFile(TEXT("LED1:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_
EXISTING,0,0);
    /*判断获取到的Handle是否是有效的,如果得到的句柄无效,则弹出错误提示窗口*/

    if(m_hled == INVALID_HANDLE_VALUE || m_hled == NULL)
    {
            MessageBox(L"Open LED Driver error!");
      }
}
```

### 5. 编写“LED1_ON”按钮程序代码

回到“Resource View”，打开程序界面，双击LED1_ON按钮。添加代码：

```
//点亮LED1灯
void CLLEDDlg::OnBnClickedButtonLed1On()
{
```

```
    int led = 1;
    DeviceIoControl(m_hled,IO_LED_ON, &led,NULL,NULL,NULL,NULL,NULL);
}//注意这里的第2和第3个参数: IO_LED_ON和led
```

### 6. 编写“LED1_OFF”按钮程序代码

同样，先给该按钮添加上点选事件的函数，然后在这个函数中添加以下程序代码：

```
//关闭LED1灯
void CLLEDDlg::OnBnClickedButtonLed1Off()
{
    int led = 1;
    DeviceIoControl(m_hled,IO_LED_OFF, &led,NULL,NULL,NULL,NULL,NULL);
}//注意这里的第2和第3个参数: IO_LED_OFF和led
```

### 7. 编写“LED2_ON”按钮程序代码

同样，先给该按钮添加上点选事件的函数，然后在这个函数中添加以下程序代码：

```
//点亮LED2灯
void CLLEDDlg::OnBnClickedButtonLed2On()
{
    int led = 2;
    DeviceIoControl(m_hled,IO_LED_ON, &led,NULL,NULL,NULL,NULL,NULL);
}//注意这里的第2和第3个参数: IO_LED_ON和led
```

### 8. 编写“LED2_OFF”按钮程序代码

先给该按钮添加上点选事件的函数，然后在这个函数中添加以下程序代码：

```
//关闭LED2灯
void CLLEDDlg::OnBnClickedButtonLed2Off()
{
    int led = 2;
    DeviceIoControl(m_hled,IO_LED_OFF, &led,NULL,NULL,NULL,NULL,NULL);
}//注意这里的第2和第3个参数: IO_LED_OFF和led
```

### 9. 编写“Close”按钮程序代码

在打开任何设备之后，如果不再使用它，应该关闭它。使用CloseHandle函数可以完成这个功能。该函数的声明如下：

```
BOOL CloseHandle(
  HANDLE hObject
);
```

该函数只有一个参数，这个参数就是打开的设备句柄，如图6-24所示。

在事件中添加代码：

```
void CLLEDDlg::OnBnClickedButtonClose()
{
    if(m_hled != NULL)
    {
            CloseHandle(m_hled);
    }
}
```

在所有代码添加完毕后，需要对程序代码进行编译、连接并生成可执行程序。

### 10. 编译源程序

在Visual Studio 2005工具栏的编译方式中选择“Release”，在SDK下拉菜单选择“DMA6410（ARMV4I）”，如图6-23所示。

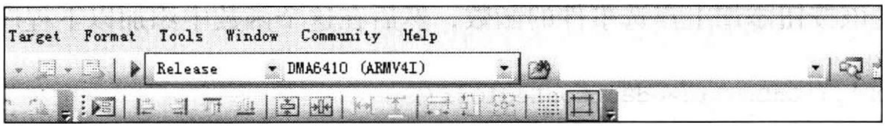



图6-23 选择Release编译方式

配置好后，依次选择“Build”→“Rebuild Led”，然后开始编译程序。编译成功后，在“Output”窗口中显示如下输出，如图6-24所示，说明编译成功。

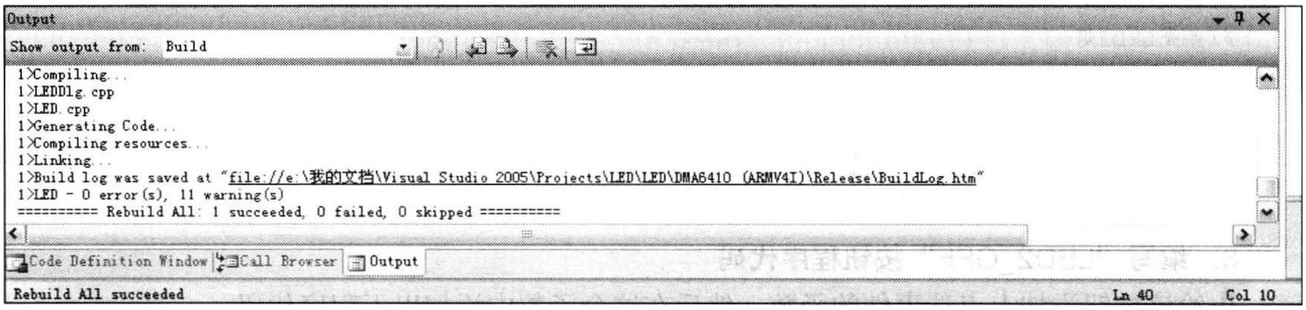



图6-24 “Output”窗口中的显示

编译成功后即生成了可执行文件LED.exe。该文件在文件夹“我的文档\Visual Studio 2005\Projects\LED\LED\DMA6410（ARMV4I）\Release”中。运行LED驱动程序时，需要将LED.exe复制到开发平台DMA-6410XP中执行。

### 11. 复制LED.exe至开发平台执行程序

使用USB线将开发平台USB_OTG接口与PC的USB口连接。打开平台开关，启动Windows Embedded CE 6.0系统，PC上的ActiveSync软件显示连接成功，双击PC右下角任务栏中的图标，打开ActiveSync软件，如图6-25所示。

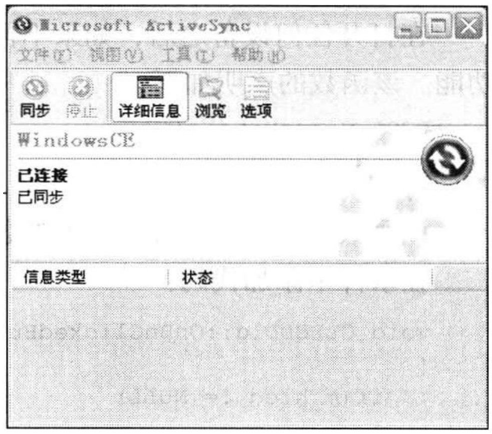



图6-25 “Microsoft ActiveSync”窗口

在ActiveSync窗口中单击“浏览”打开WINDOWS CE文件夹，找到编译好的LED.exe文件，将该文件复制到开发平台的NANDFLASH文件夹中（如图6-26所示），然后在NAND FLASH文件夹中打开LED.exe，双击运行程序。

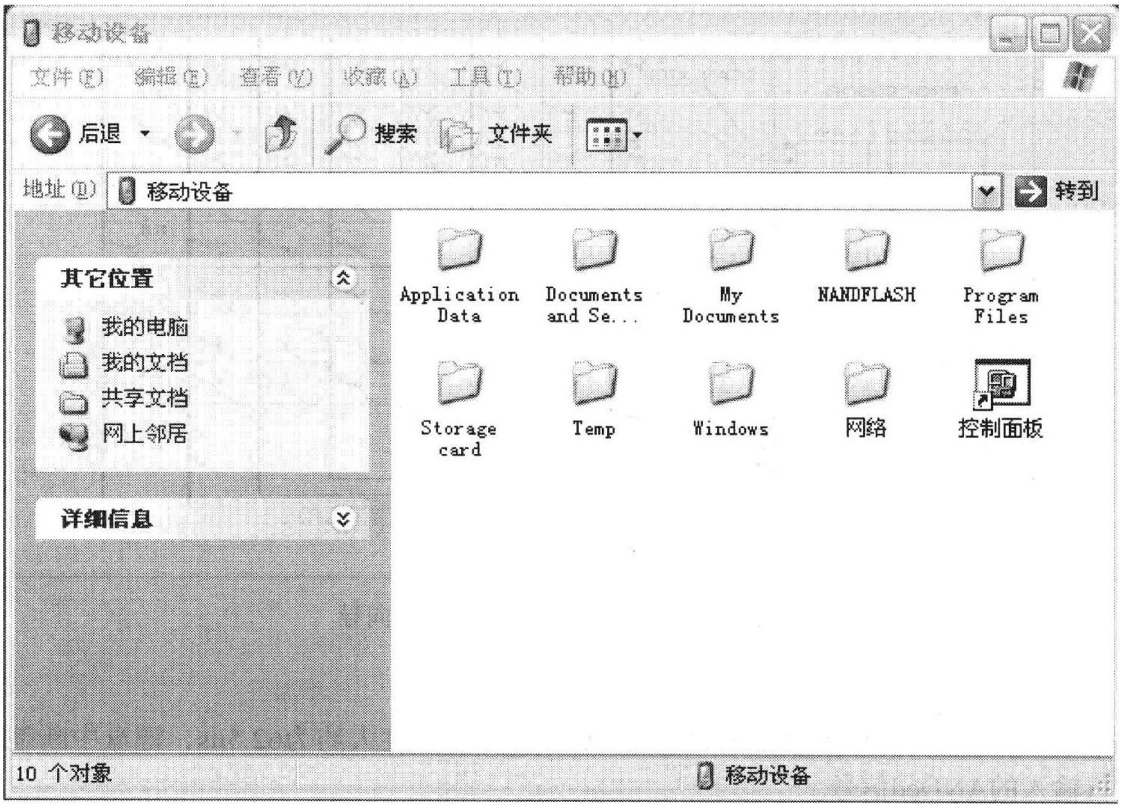



图6-26 文件复制到开发平台的NANDFLASH文件夹

在Windows Embedded CE 6.0中打开程序后，首先单击“OPEN”按钮开启LED灯控制功能，然后通过LED1_ON与LED1_OFF灯按钮控制LED灯的亮灭。

## 6.3 键盘接口及驱动程序

在DMA-6410XP实验平台上，配置有矩阵式键盘，由输入/输出端口GPIOK和GPIOL端口与矩阵式键盘接口。采用Visual Studio 2005编写矩阵式键盘驱动程序，在Windows Embedded CE 6.0下实现对矩阵式键盘扫描，并将结果显示在LCD屏幕上。

### 6.3.1 键盘概述

S3C6410内的键盘接口模块连接外部键盘设备，端口多路复用采用GPIO端口，共8行8列。CPU通过中断检测键盘按压和键盘释放事件。当行内发生任何中断时，软件用适当的程序浏览列行，检测一个或多个键盘按压及释放事件，如图6-27所示。

当键盘按压或释放，或者两种情况都发生时，提供中断状态寄存器位。为了防止开关噪音，提供内部去斗过滤器。

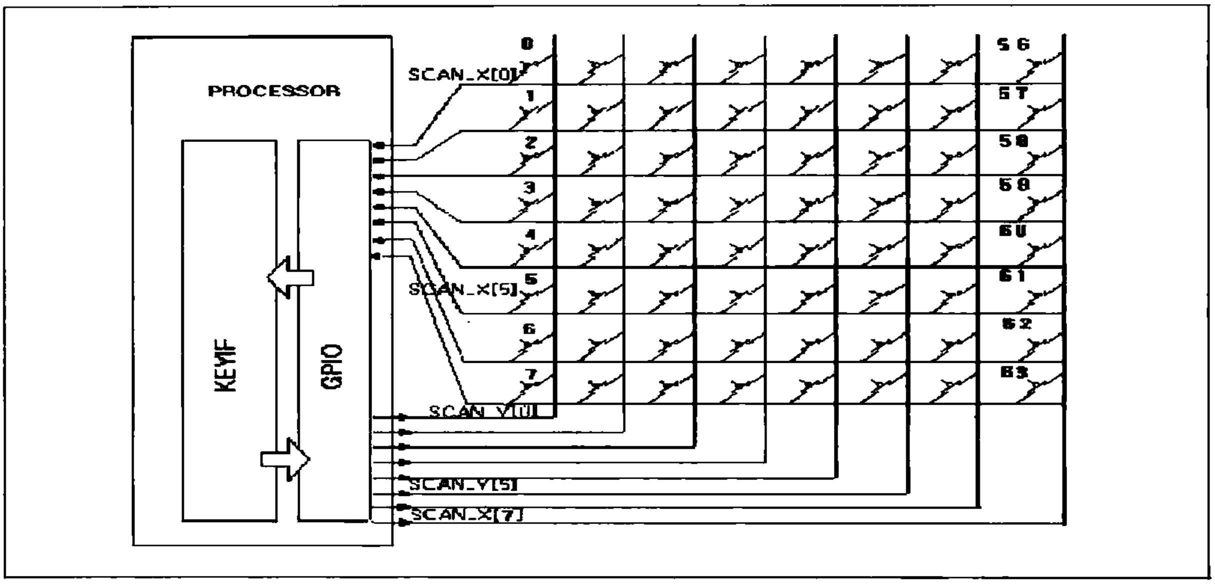



图6-27 键盘矩阵接口外部链接向导

### 1. 去斗过滤器

去斗过滤器支持任何键盘输入的键盘中断，过滤的宽度大约为62.5μs，键盘中断是过滤以后所有行输入的ANDed信号。

### 2. 滤波时钟

键盘接口去斗滤波时钟是OSC_IN，由FLT_CLK中划分出来。用户可以设置10位加计数器（KEYIFFC）的对比值。当滤波器使能位为高电平时，滤波器时钟分压器处于打开状态。FCLK的频率是FLT_CLK的功率/（（keyiffc+1）×2）。相反的，当FC_CN为低电平时，过滤时钟分压器不划分FLT_CLK。

### 3. 引脚多路复用

在S3C6410中，键盘接口输入和输出端口是GPIOK和GPIOL端口的多路复用。输入、输出端口计数由位控制，有8输入和8输出。

### 4. 唤醒源

当键盘输入作为停止模式或睡眠模式的唤醒源时，不要求KEYPADI/F寄存器的设置。唤醒需要KYEPADI/F的GPIO寄存器设置和用于屏蔽的SYSCON寄存器（PWR_CFG）。GPIO（键盘）端口1输入（GPIOK[15:8]）可以用作唤醒源，但GPIO（键盘）端口2输出（GPION[7]）不能用作唤醒源。

## 6.3.2 矩阵式键盘

矩阵式键盘由行线和列线组成，按键位于行、列的交叉点上。一个4×4的行列结构可以构成一个有16个按键的键盘。很明显，在按键数量较多的场合，矩阵式键盘与独立式键盘相比，要节省很多的I/O接口。

按键设置在行、列交叉点上，行、列分别连接到按键开关的两端。一般在行线上会通过上拉电阻连接到5V电源。在平时无键按下时，行线处于高电位状态；而当有键按下时，行线电位状态将由通过此按键的列线电位决定：列线如果为低，行线电位为低；列线如果为高，行线也为高。因为按键之间相互影响，所以必须将行线、列线信号配合起来并做适当的处理，才能确定闭合键的位置。

根据上面的描述，很容易得出按键的识别方法，分为下列的两步：

1）键盘的任意一个按键被按下，使得所有列线均为低电位，判断行线是否为低。如果行线为低，则说明该行有键按下，否则说明无键被按下。

2）如果某行有键被按下，识别键盘哪一列的键被按下，逐列置低电位，并置其余各列为高电位，检查各行线电位的变化。如果行电位变为低电位，则可确定此行此列交叉点处按键被按下。

键盘电路如图6-28所示。

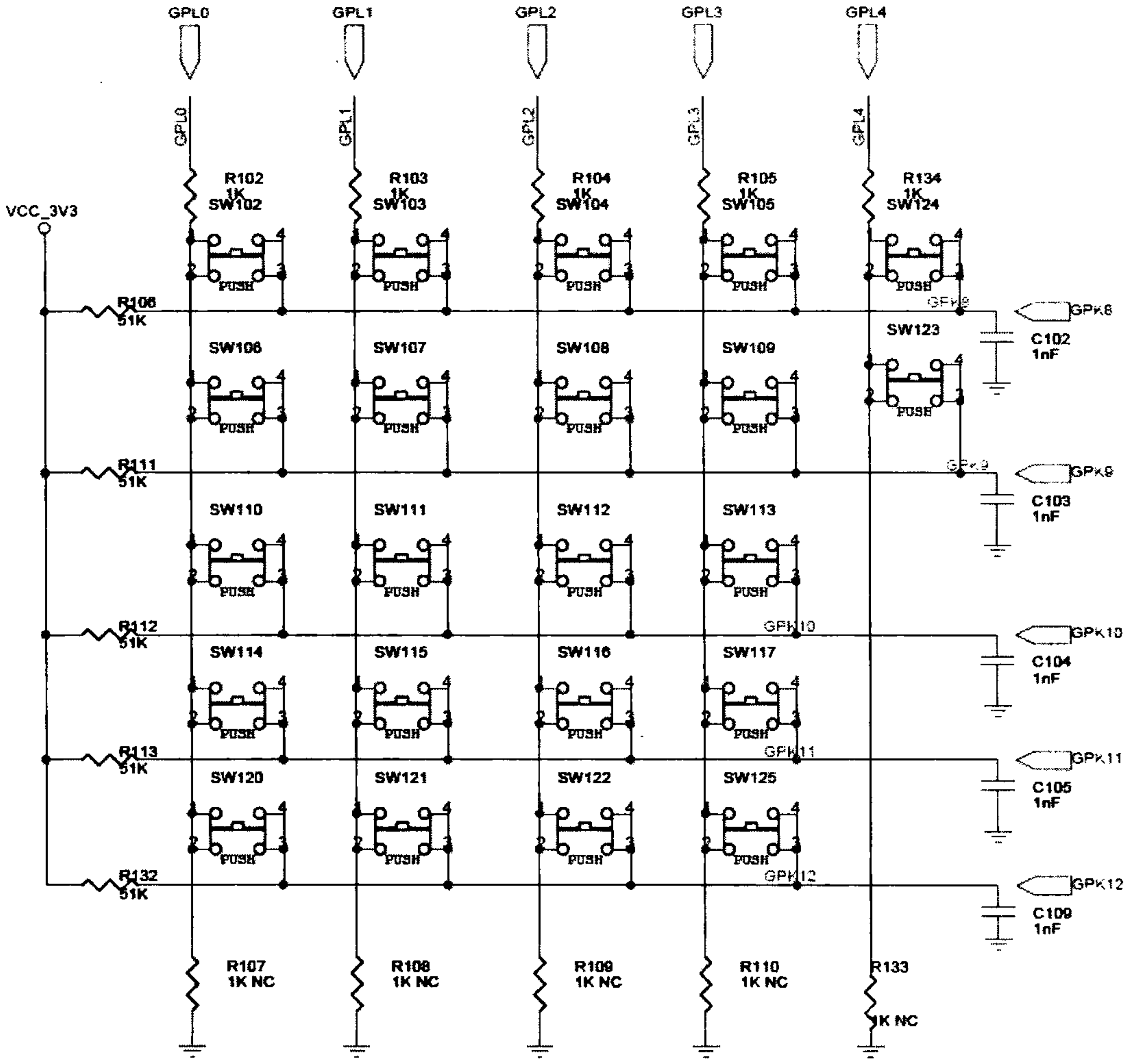


图6-28 键盘电路图

### 6.3.3 接口专用寄存器

键盘接口相关寄存器如表6-8～表6-12所示。

**表6-8 寄存器地址表**

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复 位 值 |
|---|---|---|---|---|
| GPKCON0 | 0x7F008800 | 读/写 | 端口K 配置寄存器0 | 0x22222222 |
| GPKCON1 | 0x7F008804 | 读/写 | 端口K 配置寄存器1 | 0x22222222 |
| GPKDAT | 0x7F008808 | 读/写 | 端口K 数据寄存器 | 未定义 |
| GPKPUD | 0x7F00880C | 读/写 | 端口K 上拉/下拉寄存器 | 0x55555555 |
| GPLCON0 | 0x7F008810 | 读/写 | 端口L 配置寄存器0 | 0x22222222 |
| GPLCON1 | 0x7F008814 | 读/写 | 端口L 配置寄存器1 | 0x22222222 |
| GPLDAT | 0x7F008818 | 读/写 | 端口L 数据寄存器 | 未定义 |
| GPLPUD | 0x7F00881C | 读/写 | 端口L上拉/下拉寄存器 | 0x15555555 |

**表6-9 GPKCON说明**

| GPKCON1 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| GPK8 | [3:0] | 0000=输入 0001=输出<br>0010=Host I/F DATA[8] 0011=Key pad ROW[0]<br>0100=保留 0101=DATA_CF[8]<br>0110=保留 0111=保留 | 0010 |
| GPK9 | [7:4] | 0000=输入 0001=输出<br>0010=Host I/F DATA[9] 0011= Key pad ROW[1]<br>0100=保留 0101=DATA_CF[9]<br>0110=保留 0111=保留 | 0010 |
| GPK10 | [11:8] | 0000=输入 0001=输出<br>0010=Host I/F DATA[10] 0011= Key pad ROW[2]<br>0100=保留 0101=DATA_CF[10]<br>0110=保留 0111=保留 | 0010 |
| GPK11 | [15:12] | 0000=输入 0001=输出<br>0010=Host I/F DATA[11] 0011= Key pad ROW[ 3]<br>0100=保留 0101=DATA_CF[11]<br>0110=保留 0111=保留 | 0010 |
| GPK12 | [19:16] | 0000=输入 0001=输出<br>0010=Host I/F DATA[12] 0011= Key pad ROW[4]<br>0100=保留 0101=DATA_CF[12]<br>0110=保留 0111=保留 | 0010 |

（续）

| GPKCON1 | 位 | 描 述 | | 初始状态 |
|---|---|---|---|---|
| GPK13 | [23:20] | 0000=输入<br>0010=Host I/F DATA[13]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011= Key pad ROW[5]<br>0101=DATA_CF[13]<br>0111=保留 | 0010 |
| GPK14 | [27:24] | 0000=输入<br>0010=Host I/F DATA[14]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011= Key pad ROW[6]<br>0101=DATA_CF[14]<br>0111=保留 | 0010 |
| GPK15 | [31:28] | 0000=输入<br>0010=Host I/F DATA[15]<br>0100=保留<br>0110=保留 | 0001=输出<br>0011= Key pad ROW[7]<br>0101=DATA_CF[15]<br>0111=保留 | 0010 |

表6-10 GPLCON说明

| GPLCON0 | 位 | 描 述 | | 初始状态 |
|---|---|---|---|---|
| GPL0 | [3:0] | 0000=输入<br>0010=Host I/F ADDR[0]<br>0100=保留<br>0110= ADDR_CF[0] | 0001=输出<br>0011=Key pad COL[0]<br>0101=保留<br>0111=保留 | 0010 |
| GPL1 | [7:4] | 0000=输入<br>0010=Host I/F ADDR[1]<br>0100=保留<br>0110= ADDR_CF[1] | 0001=输出<br>0011=Key pad COL[1]<br>0101=保留<br>0111=保留 | 0010 |
| GPL2 | [11:8] | 0000=输入<br>0010=Host I/F ADDR[2]<br>0100=保留<br>0110= ADDR_CF[2] | 0001=输出<br>0011=Key pad COL[2]<br>0101=保留<br>0111=保留 | 0010 |
| GPL3 | [15:12] | 0000=输入<br>0010=Host I/F ADDR[3]<br>0100=保留<br>0110= MEM0_INTata | 0001=输出<br>0011=Key pad COL[3]<br>0101=保留<br>0111=保留 | 0010 |
| GPL4 | [19:16] | 0000=输入<br>0010=Host I/F ADDR[4]<br>0100=保留<br>0110= MEM0_RESETata | 0001=输出<br>0011=Key pad COL[4]<br>0101=保留<br>0111=保留 | 0010 |
| GPL5 | [23:20] | 0000=输入<br>0010=Host I/F ADDR[5]<br>0100=保留<br>0110= MEM0_INPACKata | 0001=输出<br>0011=Key pad COL[5]<br>0101=保留<br>0111=保留 | 0010 |

（续）

| GPLCON0 | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| GPL6 | [27:24] | 0000=输入　0001=输出<br>0010=Host I/F ADDR[6]　0011=Key pad COL[6]<br>0100=保留　0101=保留<br>0110= MEM0_REGata　0111=保留 | 0010 |
| GPL7 | [31:28] | 0000=输入　0001=输出<br>0010=Host I/F ADDR[7]　0011=Key pad COL[7]<br>0100=保留　0101=保留<br>0110= MEM0_CData　0111=保留 | 0010 |

表6-11　GPKDAT说明

| GPKDAT | 位 | 描　述 |
|---|---|---|
| GPK[15:0] | [15:0] | 当端口作为输入端口时，相应的位处于引脚状态，当端口作为输出端口时，引脚状态与相应位的状态相同。当端口作为功能引脚时，读取未被定义的值 |

表6-12　GPLDAT说明

| GPLDAT | 位 | 描　述 |
|---|---|---|
| GPL[14:0] | [14:0] | 当端口作为输入端口时，相应的位处于引脚状态，当端口作为输出端口时，引脚状态与相应位的状态相同。当端口作为功能引脚时，读取未被定义的值 |

### 6.3.4 键盘驱动操作分析

Windows CE键盘驱动属于本地设备驱动。一个本地设备驱动程序会使用一些特别的接口，这种接口不同于 Windows驱动程序接口，它往往是与系统密切相关的，并且很多接口也设计为可以直接被应用程序所调用。

键盘驱动程序将键盘硬件的输入转换为驱动程序传入系统的键盘消息。驱动程序负责从这些键盘消息中产生对应的Unicode字符。键盘驱动分成两个组件，其中一些键盘驱动需要将一个按键映像为多个伪码。一个按键映像为多个虚拟键码是非常有用的。其中一些键有多个功能，或者功能有所更改，甚至是前面两者兼具。驱动程序根据某些专用按键的状态（Shift、Alt等）来生成虚拟键码。

通常，应该把键盘驱动实现为分层的驱动：

1）上层，或者称作模型设备驱动（MDD），负责将扫描码映像到虚拟键码，并且配合虚拟键码生成字符数据，然后将键盘消息打包并把它们放到系统消息队列中，这一部分是由微软公司实现的，一般不需要更改。

2）下层，或者称作依赖于平台的驱动程序（PDD），主要负责从硬件获取扫描码（DMA-6410XP矩阵键盘驱动移植主要就在这一部分）。

由如图6-29所示的键盘结构图可知，布局管理器（Layout Manager）按照下面的顺序处

理扫描码：

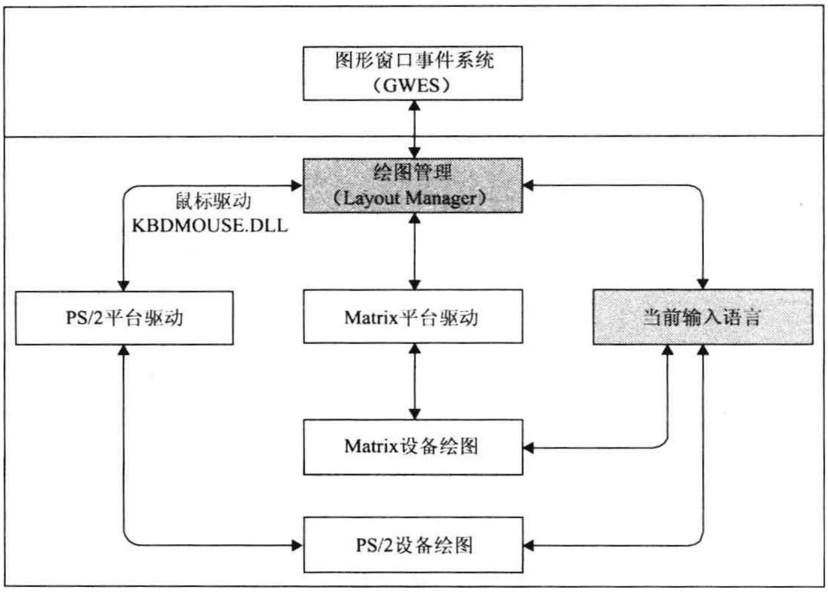



图6-29　键盘结构图

1）PDD获取一个扫描码。

2）PDD发送扫描码到布局管理器。

3）布局管理器根据发送此扫描码的键盘的设备布局将此扫描码转换为虚拟键码。

4）布局管理器根据发送此扫描码的键盘的设备布局对其进行重新映射。

5）布局管理器处理自动连击功能。所有的键盘共享自动连击设置。

6）布局管理器调用Keybd_event函数发送键盘事件。

键盘PDD是键盘驱动程序中依赖平台的程序代码。布局管理器能够同时处理多个键盘PDD，比如在同一台设备上的PS2键盘和矩阵键盘。键盘PDD中包含初始化和电源管理的函数。键盘PDD可以使用通用键盘中断服务线程（IST）或者使用自己的IST。当图形窗口和事件子系统（GWES）初始化键盘驱动时，会逐个枚举每个PDD并对其进行初始化。每个PDD都会拥有自己的入口函数，这个入口函数返回一个包含对PDD的描述和函数指针的结构体。当设备管理器初始化PDD时，键盘驱动传给PDD一个唯一的标识符，PDD使用这个标识符来标识自己。

注意，所有的PDD和布局管理器编译到一个DLL中，不能够另外添加。

在键盘驱动函数中必须指定需要初始化的PDD。以NULL作为最后一个元素的数组g_rgpfnPddEntries指定了键盘驱动中所有需要初始化的PDD。如果多个控制器使用相同的PDD，那么这个入口函数可能会被多次调用。在数组中的第一个入口点指向默认的PDD。MapVirtualKey函数使用默认的PDD设备将扫描码转换为虚拟键码。微软公司推荐将默认的PDD设置为PS/2键盘布局，以便支持HID设备。如果没有PS/2口，那么一般使用NOP PS/2 PDD，在DMA6410XP\SRC\DRIVERS\KEYBD\Pddlist\ pddlist.cpp中声明为Matrix_Entry。

Pddlist.cpp下面的程序代码声明了一个键盘PDD Matrix_Entry的入口点，并且初始化了g_rgpfnPddEntries数组。

```
        BOOL  WINAPI Matrix_Entry(
        UINT uiPddId,
        PFN_KEYBD_EVENT pfnKeybdEvent,
        PKEYBD_PDD *ppKeybdPdd
        );
    PFN_KEYBD_PDD_ENTRY g_rgpfnPddEntries[] = {
        PS2_NOP_Entry,
        Matrix_Entry,
        NULL
    };
```

当这个PDD的IST执行时，它会通过布局管理器传递下来的pfnKeybdEvent函数发送当前的键盘事件给键盘布局管理器。

\WINCE600\PUBLIC\COMMON\OAK\INC\keybdpdd.h中的程序代码说明了这个接口的使用方法。

```
     typedef struct tagKEYBD_PDD {
        WORD wPddMask;          // 键盘布局与PDD匹配
        LPCTSTR pszName;        // 用于识别PDD用户
        PFN_KEYBD_PDD_POWER_HANDLER  pfnPowerHandler;
        PFN_KEYBD_PDD_TOGGLE_LIGHTS  pfnToggleLights;
    } KEYBD_PDD, *PKEYBD_PDD;

    typedef void(*PFN_KEYBD_EVENT)(UINT uiPddId,UINT32 uiScanCode,
        BOOL fKeyDown);
```

启动键盘IST是PDD入口函数的功能之一。键盘IST调用布局管理器键盘事件函数，这个函数是由布局管理器传递给PDD的入口函数来响应每个键盘事件的。可以在驱动程序中启动并使用通用的键盘IST，只要传递给它一个包含了所有需要的信息的函数即可。

WINCE600\PLATFORM\DMA6410XP\SRC\DRIVERS\KEYBD\keypad\kbd.cpp中的相关代码如下：

```
    BOOL WINAPI Matrix_Entry(
        UINT uiPddId,
        PFN_KEYBD_EVENT  pfnKeybdEvent,
        PKEYBD_PDD *ppKeybdPdd
        )
    {
        SETFNAME(_T("Matrix_Entry"));
        BOOL fRet = FALSE;
        v_uiPddId = uiPddId;
        v_pfnKeybdEvent = pfnKeybdEvent;
    DEBUGMSG(ZONE_INIT, (_T("%s: Initialize Matrix ID %u\r\n"),pszFname,uiPddId));
        if(ppKeybdPdd == NULL)
        {
            goto leave;
        }
        *ppKeybdPdd = &MatrixPdd;
        if(Keyboard)
```

```
    {
        fRet = TRUE;
        goto leave;
    }
    //    假设有一个键盘
    Keyboard = new KeyMatrix;
    if(!KeybdDriverInitializeAddresses())
    {
        goto leave;
    }
    if(Keyboard)
    {
        Keyboard->KeybdPowerOn();
        Keyboard->IsrThreadStart();
    }
    fRet = TRUE;

leave:
DEBUGMSG(ZONE_INIT, (_T("%s: Initialization complete\r\n"),pszFname));
    return fRet;
}
```

其中，“Keyboard->KeybdPowerOn()”实现了当前的键盘布局，并设置键盘中断模式；“Keyboard->IsrThreadStart()”实现了IST的启动。实际上，DMA6410XP的IST是由IsrThreadProc（此函数实现位于WINCE600\PLATFORM\DMA6410XP\SRC\ DRIVERS\KEYBD\keypad\keymatrix.cpp）实现的（IsrThreadStart()->KBDISRThread ->IsrThreadProc）。IsrThreadProc工作过程如下：

1）等待内核的中断处理函数触发Interrupt事件。

2）收到了对键盘事件函数的调用，布局管理器开启了自动连击逻辑。

3）IST和键盘PDD并不参与自动连击功能的实现过程。

4）IST为每个返回的事件调用布局管理器的键盘事件响应函数。

5）IST调用InterruptDone结束对中断号为MATRIX_LAYOUT的中断的回应（中断号为MATRIX_LAYOUT的定义位于WINCE600\PLATFORM\DMA6410XP\SRC\INC\bsp_cfg.h）。

如果键盘很简单，只有一些数字或常用字母，需要更改的不多，如ScanCodeToVKeyTable与获取扫描码的对应，如果键盘功能复用与Matrix_0409（WINCE600\PLATFORM\DMA6410XP\SRC\DRIVERS\ KEYBD\Matrix_0409s3c6410_layout.cpp），也就是DMA6410XP用的键盘不同，那么可以增加一个自已的键盘布局或是更改Matrix_0409。

### 6.3.5 键盘底层驱动程序设计

键盘底层驱动程序如下：

```
#include <windows.h>
#include <ceddk.h>
#include <keybddbg.h>
#include <keybdpdd.h>
#include <laymgr.h>
```

```
#include "keymatrix.hpp"

void WINAPI KeybdPdd_PowerHandler(BOOL bOff);
BOOL KeybdDriverInitializeAddresses(void);

extern KeyMatrix *Keyboard;

UINT v_uiPddId;
PFN_KEYBD_EVENT v_pfnKeybdEvent;

void
ReadRegDWORD(
    LPCWSTR szKeyName,
    LPCWSTR szValueName,
    LPDWORD pdwValue
    )
{
    HKEY hKeybd;
    DWORD ValType;
    DWORD ValLen;
    DWORD status;

    // 从键的注册表中获取设备键
    status = RegOpenKeyEx(
                HKEY_LOCAL_MACHINE,
                szKeyName,
                0,
                0,
                &hKeybd);
    if(status!= ERROR_SUCCESS) {
          DEBUGMSG(ZONE_ERROR, (TEXT("ReadRegDWORD: RegOpenKeyEx(HLM\\%s)returned
%d!!!\r\n"),szKeyName,status));
        *pdwValue = 0;   // 没有获取设备键
        return;
    }

    ValLen = sizeof(DWORD);
    status = RegQueryValueEx(       // 检索值
                hKeybd,
                szValueName,
                NULL,
                &ValType,
                (PUCHAR)pdwValue,
                &ValLen);
    if(status != ERROR_SUCCESS) {
          DEBUGMSG(ZONE_ERROR, (TEXT("ReadRegDWORD: RegQueryValueEx(%s)returned
%d\r\n"),
                  szValueName,status));
        *pdwValue = 0;
    }
```

```
        DEBUGMSG(ZONE_ERROR, (_T("ReadRegDWORD(): %s -> %s is 0x%x\r\n"),szKeyName,
szValueName, *pdwValue));

        RegCloseKey(hKeybd);
    }   // 打开设备键

    static void WINAPI Matrix_PowerHandler(UINT uiPddId,BOOL fTurnOff)
    {
       KeybdPdd_PowerHandler(fTurnOff);
    }

    static void WINAPI Matrix_ToggleLights(UINT uiPddId,KEY_STATE_FLAGS KeyStateFlags)
    {
        SETFNAME(_T("Matrix_ToggleLights"));

        static const KEY_STATE_FLAGS ksfLightMask = KeyShiftCapitalFlag |
        KeyShiftNumLockFlag | KeyShiftScrollLockFlag;
        static KEY_STATE_FLAGS ksfCurr;

        KEY_STATE_FLAGS ksfNewState = (ksfLightMask & KeyStateFlags);

        if(ksfNewState != ksfCurr)
        {
          DEBUGMSG(ZONE_PDD, (_T("%s: PDD %u: Changing light state\r\n"),pszFname,uiPddId));
          KeybdPdd_ToggleKeyNotification(ksfNewState);
          ksfCurr = ksfNewState;
        }

        return;
    }

    static KEYBD_PDD MatrixPdd =
    {
        MATRIX_PDD,
        _T("Matrix"),
        Matrix_PowerHandler,
        Matrix_ToggleLights
    };

    BOOL
    WINAPI
    Matrix_Entry(
        UINT uiPddId,
        PFN_KEYBD_EVENT pfnKeybdEvent,
        PKEYBD_PDD *ppKeybdPdd
        )
    {
        SETFNAME(_T("Matrix_Entry"));

        BOOL fRet = FALSE;

        v_uiPddId = uiPddId;
```

```
    v_pfnKeybdEvent = pfnKeybdEvent;

    DEBUGMSG(ZONE_INIT, (_T("%s: Initialize Matrix ID %u\r\n"),pszFname,uiPddId));

    if(ppKeybdPdd == NULL)
    {
        goto leave;
    }

    *ppKeybdPdd = &MatrixPdd;

    if(Keyboard)
    {
        fRet = TRUE;
        goto leave;
    }

        //假设有一个键盘
    Keyboard = new KeyMatrix;

    if(!KeybdDriverInitializeAddresses())
    {
        goto leave;
    }

    if(Keyboard)
    {
        Keyboard->KeybdPowerOn();
        Keyboard->IsrThreadStart();
    }

    fRet = TRUE;

leave:
    DEBUGMSG(ZONE_INIT, (_T("%s: Initialization complete\r\n"),pszFname));
    return fRet;
}
#ifdef DEBUG
// 验证函数声明与定义
static PFN_KEYBD_PDD_ENTRY v_pfnKeybdEntry = Matrix_Entry;
#endif
```

## 6.3.6 键盘驱动程序设计

### 1. 建立Keypad工程

首先启动Visual Studio 2005，选择“File”→“New”→“Project”，弹出相应对话框，在左侧“Project types：”中选择Visual C++下的“Smart Device”，在右侧选择“MFC Smart Device Application”，下方“Name”改为“Keypad”，然后单击“OK”按钮，开始建立工程，接着单击“Next”按钮，在“Platforms”属性中选择DMA-6410的SDK加入到“Selected

SDKs”中，并移除“Selected SDKs”中的“Pocket PC 2003”，完成后单击“Next”按钮，进入下一步。

在“Application Type”界面的“Applicatoin Type”中选择“Dialog based”，在“Use of MFC”中选择“Use MFC in a static library”，在“Resource language”下拉列表框中选择“中文（中国）”。完成后单击“Next”按钮，进入下一步。

接下来的步骤依照默认选择即可，最后单击“Finish”按钮完成工程的建立。

### 2. 设计界面

首先，依次选择“View”→“Resource View”，然后在“Resource View”中依次展开“Keypad”→“Keypad.rc”→“Dialog”，随后双击IDD_KEYPAD_DIALOG，出现如图6-30所示的界面。驱动键盘只需要一个空白的界面，通过激发键盘点击事件弹出消息窗口的方式来验证键盘驱动程序。

下面要做的是实现键盘的响应功能。若按下键盘的“0”键，则显示一则消息框，提示“Press Key 0”。

### 3. 重写PreTranslateMessage函数

在PreTranslateMessage函数中可以处理窗口收到的任何消息，该函数在MFC框架的程序中已经实现了，只是IDE环境隐藏了它，没有把它显示出来。

首先，在键盘上按下<Alt+Ctrl>组合键，把属性窗口显示出来，接着依次选择菜单项“View”→“Solution Explorer”，在该窗口中依次展开“Key”→“Header Files”，然后双击“KeyDlg.h”文件，随后在CKeypadDlg类的任意位置单击一下，并在右下角属性视窗中单击图标按钮，切换到重写函数列表，在PreTranslateMessage栏右侧的选择列表中选择“<Add> PreTranslateMessage”，如图6-31所示。

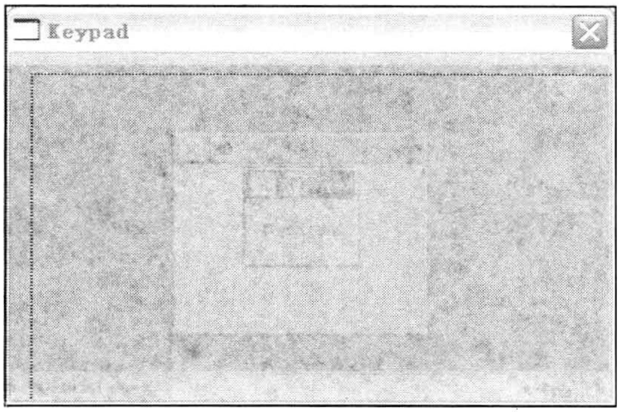



图6-30 驱动键盘的界面

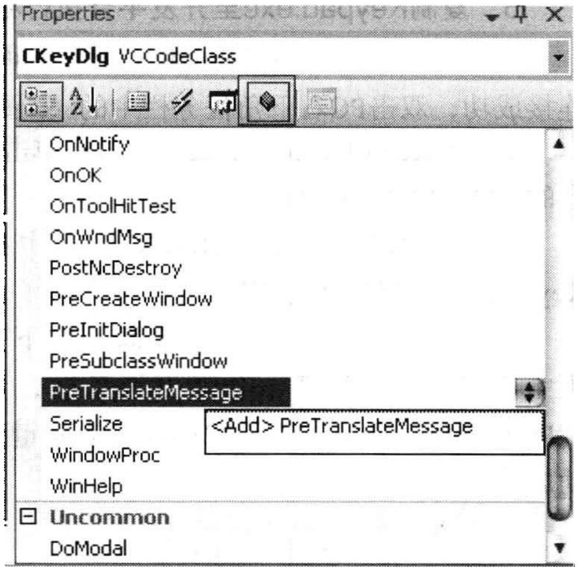



图6-31 属性窗口

这样，可以在“KeyDlg.cpp”文件中看到如下函数：

```
BOOL CKeyDlg::PreTranslateMessage(MSG* pMsg)
{
        return CDialog::PreTranslateMessage(pMsg);
}
```

注意，在return语句前添加键盘消息处理程序代码即可。

### 4. 添加键盘消息处理程序代码

PreTranslateMessage函数有一个MSG类型的指标参数。MSG是一个消息类型的结构体，并不需要了解它的全部结构，只需要知道如何判断按键消息即可。在该函数中添加如下的程序代码：

```
if(pMsg->message == WM_KEYDOWN)                    //判断消息是否为键盘按下的消息
   {
        if(pMsg->wParam == '0')                     //判断按下的是否是0键
        {
             MessageBox(_T("Press Key 0"));         //弹出消息提示框
        }
   }
```

### 5. 编译测试

在Visual Studio 2005上方工具栏中间的编译方式中选择“Release”，在SDK下拉菜单中选择“DMA6410（ARMV4I）”。

配置好后，依次选择“Build”→“Rebuild Keypad”，开始编译程序。编译成功后即生成了可执行文件Keypad.exe，该文件在文件夹“我的文档\Visual Studio 2005\Projects\Keypad\Keypad\DMA6410（ARMV4I）\Release”中。为了验证Keypad驱动程序是否正确，将Keypad.exe复制到开发平台DMA-6410XP中。

### 6. 复制Keypad.exe至开发平台执行程序

打开平台开关，启动Windows Embedded CE 6.0系统。启动后，PC上的ActiveSync软件显示连接成功，双击PC右下角任务栏中的图标，打开ActiveSync软件。

在“Microsoft ActiveSync”窗口中单击“浏览”打开开发平台文件夹，将文件复制到开发平台的NANDFLASH文件夹。

然后在NAND FLASH文件夹中打开Keypad.exe，双击运行程序。

打开程序后是一个空白界面。按下开发板上键盘的“0”键，会弹出消息框，显示“Press Key 0”，如图6-32所示，表示键盘驱动成功。

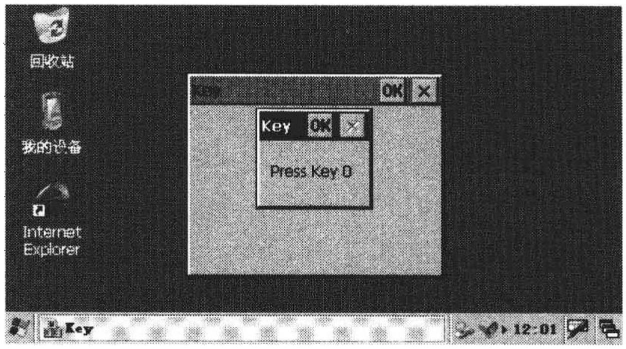



图6-32 消息框显示“Press Key 0”

### 7. 编写LED控制快捷键程序

在该程序中，在界面载入时即开启LED控

制功能。用来控制LED灯亮和灭的快捷键依次如下。

- 0：打开LED1。
- 1：关闭LED1。
- 2：打开LED2。
- 3：关闭LED2。

接下来添加相应代码。

在KeypadDlg.h文件中添加LED控制句柄，代码如下：

```
HANDLE m_hled;    //设备控制码
```

在KeypadDlg.cpp中添加LED驱动代码，如下：

```
typedef enum _LED_IO_CONTROL_VALUE
{
   IO_LED_ON = 0x10,
   IO_LED_OFF = 0x11,
}ENUM_LED_IO_CONTROL, *PENUM_LED_IO_CONTROL;
```

在函数PreTranslateMessage下直接添加开启LED控制功能的代码。

```
m_hled = CreateFile(TEXT("LED1:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_
EXISTING,0,0);
if(m_hled == INVALID_HANDLE_VALUE || m_hled == NULL)
{
   MessageBox(L"Open LED DRIVER ERROR");
}
```

在判断键盘动作的代码中添加如下内容。

```
if(pMsg->message == WM_KEYDOWN)  //判断消息是否为键盘按下的消息
{
   if(pMsg->wParam == '0')       //判断按下的是否是0键
   {
         int led = 1;
         DeviceIoControl(m_hled,IO_LED_ON, &led,NULL,NULL,NULL,NULL,NULL);
   }

   if(pMsg->wParam == '1')              //判断按下的是否是1键
   {
         int led = 1;
         DeviceIoControl(m_hled,IO_LED_OFF, &led,NULL,NULL,NULL,NULL,NULL);
   }
   if(pMsg->wParam == '2')       //判断按下的是否是2键
   {
         int led = 2;
         DeviceIoControl(m_hled,IO_LED_ON, &led,NULL,NULL,NULL,NULL,NULL);
   }
   if(pMsg->wParam == '3')       //判断按下的是否是3键
   {
         int led = 2;
         DeviceIoControl(m_hled,IO_LED_OFF, &led,NULL,NULL,NULL,NULL,NULL);
   }
}
```

添加代码后的PreTranslateMessage函数如图6-33所示。

```
BOOL CKeypadDlg::PreTranslateMessage(MSG* pMsg)
{
    // TODO: Add your specialized code here and/or call the base class
    m_hled = CreateFile(TEXT("LED1:"), GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,0,0);
    if(m_hled == INVALID_HANDLE_VALUE || m_hled == NULL)
    {
        MessageBox(L"Open LED DRIVER ERROR");
    }
    if(pMsg->message == WM_KEYDOWN) //判断消息是否为键盘按下的消息
    {
        if(pMsg->wParam == '0')     //判断按下的是否是0键
        {
            int led = 1;
            DeviceIoControl(m_hled, IO_LED_ON, &led, NULL, NULL, NULL, NULL,NULL);
        }
        if(pMsg->wParam == '1')     //判断按下的是否是1键
        {
            int led = 1;
            DeviceIoControl(m_hled, IO_LED_OFF, &led, NULL, NULL, NULL,NULL,NULL);
        }

        if(pMsg->wParam == '2')     //判断按下的是否是2键
        {
            int led = 2;
            DeviceIoControl(m_hled, IO_LED_ON, &led, NULL, NULL, NULL, NULL, NULL);
        }
        if(pMsg->wParam == '3')     //判断按下的是否是3键
        {
            int led = 2;
            DeviceIoControl(m_hled, IO_LED_OFF, &led, NULL, NULL, NULL,NULL,NULL);
        }
    }
    return CDialog::PreTranslateMessage(pMsg);
}
```

图6-33 添加代码后的PreTranslateMessage函数

完成代码的编写后重复执行上面的第5步和第6步，对源代码进行编译，建立并将生成的Keypad.exe文件复制到开发平台覆盖之前的Keypad文件后执行。在开发平台打开Keypad程序后，按下“0”、“1”、“2”、“3”键可分别对两个LED灯进行开或关操作。

## 6.4 ADC接口及驱动程序

### 6.4.1 S3C6410 ADC

S3C6410微处理器上设有10位CMOS的ADC（模/数转换器），它是一种循环类型的模/数转换器，具有8位通道模拟输入。它将模拟的输入信号转换成10位二进制数字编码，最大转换率是500KSPS和2.5MHz的ADC时钟。ADC的操作带有片上采样保持功能，具有电源中断模式的支持。触摸屏接口控制触摸屏的位置和方位（XP，XM，YP，YM）为X坐标转换和Y坐标转换选择触摸屏的位置和方位（XP，XM，YP，YM）。触摸屏界面包含了位置和方位控制逻辑、ADC界面逻辑和中断发生逻辑。

ADC及触摸屏接口包括以下功能及特性。

- 分辨率：10位。
- 微分线性误差：±1.0LSB。
- 积分线性误差：±2.0LSB。
- 最高转换率：500KSPS。
- 低耗电量。
- 供电电压：3.3V。
- 模拟输入范围：0～3.3V。
- 对芯片采样保持功能。
- 正常转换模式。
- 单独的X/Y坐标的转换模式。
- 自动（顺序）的X/Y坐标的转换模式。
- 等待中断方式。
- 停止模式唤醒源。

## 6.4.2 A/D转换接口电路

在DMA-6410XP实验平台上，一个温度传感器LM35D连接到A/D转换器的0通道，3个模拟信号连接到A/D转换器的1、2和3通道。采用Visual Studio 2005编写A/D转换程序，在Windows Embedded CE 6.0下实现对4个模拟量的A/D转换，并将结果显示在LCD屏幕上。

### 1. S3C6410的A/D转换器

ADC和触摸屏接口的功能结构框图，如图6-34所示。

当触摸屏装置使用时，触摸屏的I/F、XM或YM只接地。当触摸屏的装置未使用时，为正常ADC转换，XM或YM连接模拟输入信号。

（1）A/D转换时间

当GCLK频率是50MHz，分频器值是49时，总的10位转换时间如下：

$$A/D转换频率=50MHz/(49+1)=1MHz$$

$$转换时间=1/(1MHz/5周期)=1/200kHz=5\mu s$$

ADC可在最高为2.5MHz时钟下操作，因此，转换率可高达500KSPS。

（2）触摸屏接口方式

1）正常转换模式。

单个转换模式用于通用的ADC转换。这种模式可以通过设置ADCCON（ADC的控制寄存器）初始化，并完成读和写存入ADCAT0（ADC数据寄存器0）。

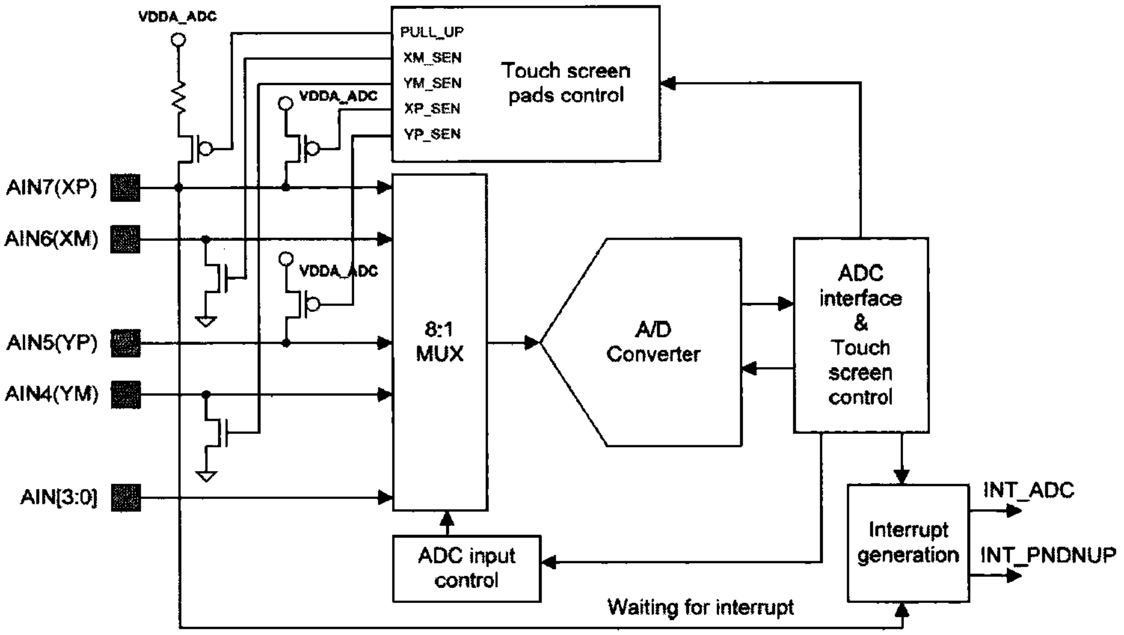



图6-34 ADC和触摸屏接口的功能结构框图

2）单独的X/Y坐标转换模式。

触摸屏控制器可以使用两个转换模式中的一个进行转换。单独的X/Y坐标转换模式可以在以下方法中转换：X坐标模式写X坐标的转换数据到ADCDAT0，因此，触摸屏接口产生中断源到中断控制器。Y坐标模式写Y坐标的转换数据到ADCDAT1，因此，触摸屏接口生成中断源到中断控制器。

3）自动（顺序）的X/Y坐标转换模式。

自动（顺序）的X/Y坐标转换模式可以在以下方法中转换：触摸屏控制器顺序转换X坐标和Y坐标。当触摸屏被触摸时，X测量数据写到ADCDAT0数据寄存器，Y测量数据写到ADCDAT1数据寄存器。触摸屏接口在自动位置转换模式上，产生中断源到中断控制器。

4）等待中断方式。

当该系统在停止模式（电源中断）时，触摸屏控制器产生唤醒信号（WKU）。在触摸屏接口下，触摸屏控制器等待中断模式必须设置位置和方位状态（XP，XM，YP，YM）。在触摸屏控制器产生唤醒信号（Wake-Up）后，等待中断方式必须清除。

（3）待机模式

当ADCCON[2]设定为“1”时，待机模式被激活。在此模式下，A/D转换操作停止，并且ADCDAT0和ADCDAT1寄存器包含先前转换的数据。

### 2. S3C6410的A/D转换接口电路

S3C6410的A/D转换接口电路如图6-35所示。

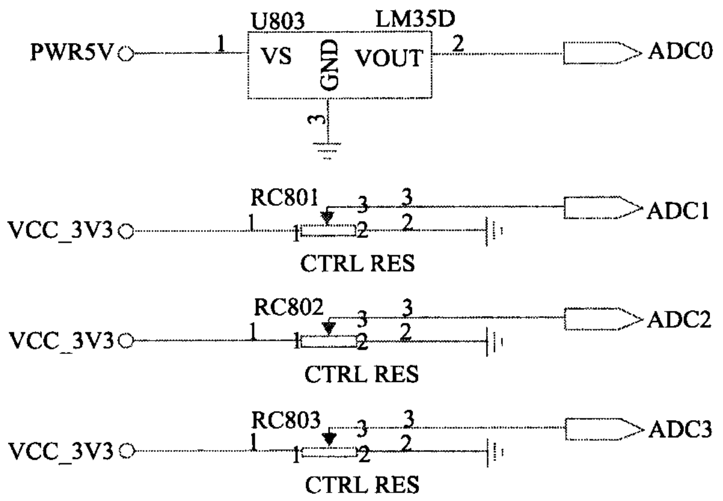



图6-35　S3C6410的A/D转换接口电路

## 6.4.3　ADC专用寄存器

下面介绍ADC的专用寄存器。

### 1. ADC控制寄存器（ADCCON）

ADC控制寄存器及其位分配分别如表6-13和表6-14所示。

表6-13　ADC控制寄存器

| 寄 存 器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| ADCCON | 0x7E00B000 | 读/写 | ADC控制寄存器 | 0x3FC4 |

表6-14　ADC控制寄存器的位分配

| ADCCON | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| ECFLG | [15] | 转换的结束标记（只读）<br>0=正在进行A/D转换<br>1= A/D转换结束 | 0 |
| PRSCEN | [14] | ADC预定标器启动。<br>0 =禁用<br>1 =启动 | 0 |
| PRSCVL | [13:6] | ADC预定标器值0xFF<br>数据值：5～255 | 0xFF |

（续）

| ADCCON | 位 | 描 述 | 初始状态 |
| --- | --- | --- | --- |
| SEL_MUX | [5:3] | 模拟输入通道选择<br>000 = AIN 0<br>001 = AIN 1<br>010 = AIN 2<br>011 = AIN 3<br>100 = YM<br>101 = YP<br>110 = XM<br>111 = XP | 0 |
| STDBM | [2] | 待机模式选择<br>0 =正常运作模式<br>1 =待机模式 | 1 |
| READ_START | [1] | A/D转换开始读取<br>0 =禁用开始读操作<br>1 =启动开始读操作 | 0 |
| ENABLE_START | [0] | A/D转换开始启用<br>如果READ_START启用，这个值是无效的<br>0 =无行动<br>1 =A/D转换开始和该位被清理后开启 | 0 |

## 2. ADC触摸屏控制寄存器（ADCTSC）

ADC触摸屏控制寄存器及其位分配分别如表6-15和表6-16所示。

表6-15 ADC触摸屏控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
| --- | --- | --- | --- | --- |
| ADCTSC | 0x7E00B004 | 读/写 | ADC触摸屏控制寄存器 | 0x58 |

表6-16 ADC触摸屏控制寄存器的位分配

| ADCTSC | 位 | 描 述 | 初始状态 |
| --- | --- | --- | --- |
| UD_SEN | [8] | 检测触摸笔向上或向下的位置<br>0 =触摸笔向下中断信号 | 0 |
| YM_SEN | [7] | YM开关启动<br>0 =YM输出驱动器禁用<br>1 =YM输出驱动器启动 | 0 |
| YP_SEN | [6] | YP开关启动<br>0 =YP输出驱动器禁用<br>1 =YP输出驱动器启动 | 1 |

（续）

| ADCTSC | 位 | 描述 | 初始状态 |
|---|---|---|---|
| XM_SEN | [5] | XM开关启动<br>0 =XM输出驱动器禁用<br>1 =XM输出驱动器启动 | 0 |
| XP_SEN | [4] | XP开关启动<br>0 =XP输出驱动器禁用<br>1 =XP输出驱动器启动 | 1 |
| PULL_UP | [3] | 上拉开关启动<br>0 =XP上拉启用<br>1 =XP上拉禁用 | 1 |
| AUTO_PST | [2] | X和Y位置的自动定序转换<br>0 =正常的ADC转换<br>1 =X和Y位置的自动定序测量 | 0 |
| XY_PST | [1:0] | X和Y坐标的手动测量<br>00 =没有运作模式<br>01 = X坐标测量<br>10 = Y坐标测量<br>11 =等待中断方式 | 0 |

注意以下两点：

1）等待触摸屏中断时，XP_SEN位必须设置为“1”，即“XP输出启动”，PULL_UP位必须设置为“0”，即“XP上拉启动”。

2）在自动定序的X/Y坐标转换下，AUTO_PST应设置为“1”。

### 3. X/Y坐标转换下触摸屏的PIN条件

X/Y坐标转换下触摸屏的PIN条件如表6-17所示。

表6-17 X/Y坐标转换下触摸屏的PIN条件

| | XP | XM | YP | YM | ADC通道选择 |
|---|---|---|---|---|---|
| X坐标 | Vref | GUN | Hi-Z | Hi-Z | YP |
| Y坐标 | Hi-Z | Hi-Z | Vref | GND | XP |

### 4. ADC启动延迟寄存器（ADCDLY）

ADC启动延迟寄存器及其位分配分别如表6-18和表6-19所示。

表6-18 ADC启动延迟寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| ADCDLY | 0x7E00B008 | 读/写 | ADC启动或时间延迟寄存器 | 0x00ff |

表6-19 ADC启动延迟寄存器的位分配

| ADCDLY | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| FILCLKsrc | [16] | ADCDLY时钟初始化。<br>0 =外部输入时钟<br>1 =RTC时钟 | 0 |
| DELAY | [15:0] | 1）在ADC转换模式（正常、单独和自动转换）下，通过计算此值使ADC的转换延迟。计数时钟PCLK得到ADC转换延迟值<br>2）在等待中断模式下，当触摸笔产生中断信号（INT_PNDNOP）时，几毫秒的时间自动完成X/Y位置转换。如果中断发生在停止模式下，它产生唤醒信号（若干毫秒），退出停止模式 | 00ff |

### 5. ADC数据转换寄存器0（ADCDAT0）

ADC数据转换寄存器0及其位分配分别如表6-20和表6-21所示。

表6-20 ADC数据转换寄存器0

| 寄 存 器 | 地　址 | 读/写 | 描　述 | 复位值 |
|---|---|---|---|---|
| ADCDAT0 | 0x7E00B00c | 读 | ADC数据转换寄存器0 | |

表6-21 ADC数据转换寄存器0的位分配

| ADCDAT0 | 位 | 描　述 | 初始状态 |
|---|---|---|---|
| UPDOWM | [15] | 在等待中断模式下，触摸笔向上或向下状态<br>0 =触摸笔向下状态<br>1 =触摸笔向上状态 | |
| AUTO_PST | [14] | X和Y坐标的自动定序转换<br>0 =正常的ADC转换<br>1 =X和Y坐标的定序测量 | |
| XY_PST | [13:12] | X和Y坐标的手动测量<br>00 =没有运作模式<br>01 =X的坐标测量<br>10 =Y的坐标测量<br>11=等待中断方式 | |
| Reserved | [11:10] | 保留 | |
| XPDATA<br>（Normal ADC） | [9:0] | X坐标的数据转换（包括正常ADC的转换数据值）<br>数据值：0x0～0x3ff | |

### 6. ADC数据转换寄存器1（ADCDAT1）

ADC数据转换寄存器1及其位分配分别如表6-22和表6-23所示。

表6-22 ADC数据转换寄存器1

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| ADCDAT1 | 0x7E00B010 | 读 | ADC数据转换寄存器1 | |

表6-23 ADC数据转换寄存器1的位分配

| ADCDAT1 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| UPDOWM | [15] | 在等待中断模式下，触摸笔向上或向下状态<br>0 =触摸笔向下状态<br>1 =触摸笔向上状态 | |
| AUTO_PST | [14] | X和Y坐标的自动定序转换<br>0 =正常的ADC转换<br>1 =X和Y坐标的定序测量 | |
| XY_PST | [13:12] | X和Y坐标的手动测量<br>00 =没有运作模式<br>01 =X的坐标测量<br>10 =Y的坐标测量<br>11=等待中断方式 | |
| Reserved | [11:10] | 保留 | |
| XPDATA<br>（Normal ADC） | [9:0] | Y坐标的数据转换<br>数据值：0x0～0x3ff | |

### 7. ADC触摸屏中断清除寄存器

当中断服务完成后，中断服务程序负责清理中断，在ADL触摸屏中断清除寄存器中写入任何数据都将清除相关有效的中断。当它被读取时，未定义的值将返回。 该寄存器有两种，即ADCCLRINT和ADCCLRWK，分别如表6-24和表6-25所示。

表6-24 ADCCLRINT

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| ADCCLRINT | 0x7E00B018 | 写 | 清除ADC中断 | |

表6-25 ADCCLRWK

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| ADCCLRWK | 0x7E00B020 | 写 | 清除唤醒中断 | |

## 6.4.4 A/D转换驱动分析

A/D转换一般具备以下功能：

- 初始化设备——调用ADC_Init。
- 设置相关参数并读取转换结果——ADC_Read。

ADC驱动的操作流程图如图6-36所示。

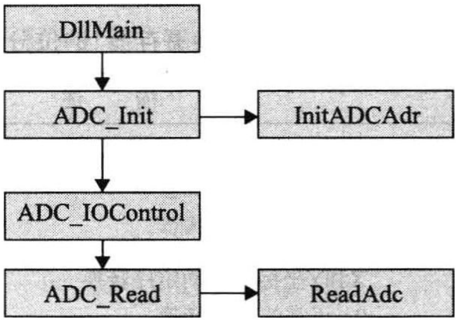



图6-36 ADC驱动的操作流程图

DllMain()为流驱动入口函数。ADC_Init()函数调用InitADCAdr()中的MmMapIoSpace()函数完成ADC物理地址到操作系统虚拟地址的转换，对虚拟地址空间的操作就相当于对ADC寄存器的物理地址操作。ADC_Deinit()用于驱动卸载。ADC_IOControl()函数接收上层应用传送过来的命令字和参数，如ADC打开（IO_ADC_ON）、ADC关闭（IO_ADC_OFF）、通道选择（IO_ADC_SEL_ CHANNEL）和频率选择（IO_ADC_SEL_FREQ）。ADC_Read()函数调用Read_Adc()函数进行A/D转换，并读取寄存器ADCDAT0的数据。

## 6.4.5 A/D转换底层驱动程序设计

```
#include <windows.h>
#include <s3c6410.h>
#include <ceddk.h>
#include "s3c6410_adc.h"
#include "soc_cfg.h"

#define ADC_DBG_MSG 0
#define ADC_ERR_MSG 0

#define ADC_FREQ 2500000
#define LOOP 10000
#define S3C6410_BASE_REG_PA_ADC   0x7E00B000

typedef enum _ADC_IO_CONTROL_VALUE {
    IO_ADC_ON = 0x40,
    IO_ADC_OFF = 0x41,
    IO_ADC_SEL_CHANNEL,
    IO_ADC_SEL_FREQ
} ENUM_ADC_IO_CONTROL, *PENUM_ADC_IO_CONTROL;

struct Adc_Para
{
   int   nChannel;
```

```
    int   nPrescaler;
    BOOL bOn;
}g_adcPara;

// ADC寄存器虚拟地址
volatile S3C6410_ADC_REG* v_pADC_Regs = NULL;
volatile S3C6410_GPIO_REG *v_pIOP_ADC_regs = NULL;

volatile UINT g_uPreScaler;


int AdcData[4] = {-1};    //新增的4个通道的ADC转换结果保存,并返回读取

BOOL WINAPI DllMain(HANDLE hinstDll,ULONG Reason,LPVOID Reserved)
{
   switch(Reason)
   {
          case DLL_PROCESS_ATTACH:
                 RETAILMSG(ADC_DBG_MSG,(TEXT("S3C6410 ADC: DLL_PROCESS_ATTCH\r\n")));
                 break;

          case DLL_PROCESS_DETACH:
                 RETAILMSG(ADC_DBG_MSG,(TEXT("S3C6410 ADC: DLL_PROCESS_DETACH\r\n")));
                 break;

          default:
                 break;
   }

   return TRUE;
}


BOOL InitADCAdr()
{
   DWORD dwIOSize;

   PHYSICAL_ADDRESS  IOPhyADC = {S3C6410_BASE_REG_PA_ADC,0};
   RETAILMSG(ADC_DBG_MSG, (TEXT("++InitADCAdr\r\n")));

   dwIOSize = sizeof(S3C6410_ADC_REG);

   if(NULL == v_pADC_Regs)
   {
          v_pADC_Regs  =  (volatile  S3C6410_ADC_REG  *)MmMapIoSpace(IOPhyADC,
(ULONG)dwIOSize,FALSE);

          if(NULL == v_pADC_Regs)
          {
                 RETAILMSG(ADC_ERR_MSG, (TEXT("--InitADCAdr failen !\r\n")));
                 return FALSE;
          }
   }
```

```
    RETAILMSG(ADC_DBG_MSG, (TEXT("--InitADCAdr success\r\n")));

    return TRUE;
}

int ReadAdc(int nChannel,int PreScaler)
{
    static int nPreChannel = -1;
    int i,j=0;
    //10位A/D转换,启动A/D预分频器,设置A/D预分频器的值
    v_pADC_Regs->ADCCON &= ~(0xffff);
    v_pADC_Regs->ADCCON = (0<<16)|(1<<14)|(PreScaler<<6);

    if(nPreChannel != nChannel)           //选择ADC通道
    {
            nPreChannel = nChannel;
    }
    v_pADC_Regs->ADCCON &= ~(0x7<<3);
    v_pADC_Regs->ADCCON |= (nPreChannel<<3);
    for(i =0 ; i< LOOP; i++);             // 延迟设置通道
    RETAILMSG(ADC_DBG_MSG, (TEXT("channel = d%d \r\n"), (v_pADC_Regs->ADCCON>>3)&0x07));

    v_pADC_Regs->ADCCON |= 0x1;           // 启动ADC
    //ADC在启动A/D转换同时启动位被清零
    while(v_pADC_Regs->ADCCON & 0x1);     // 检查启动位是否为低,它将自动清零

    //检查结束标志是否为高
    while(!(v_pADC_Regs->ADCCON & 0x8000));

    AdcData[nPreChannel] = v_pADC_Regs->ADCDAT0 & 0x3ff;

#if 0//for degug
    //RETAILMSG(ADC_DBG_MSG, (TEXT("AdcData[%d] = %d\r\n"),nChannel,AdcData[nChannel]));
    //RETAILMSG(ADC_DBG_MSG, (TEXT("ReadAdc---\r\n")));
#endif

    return(AdcData[nPreChannel]);
}

//ADC_IOControl函数
BOOL ADC_IOControl(DWORD dwOpenContext,DWORD dwIoControlCode,LPBYTE lpInBuf,
                   DWORD nInBufSize,LPBYTE lpOutBuf,DWORD nOutBufSize,
                   LPDWORD lpBytesReturned)
{
    DWORD dwFreq;

    RETAILMSG(ADC_DBG_MSG,(TEXT("dwIoControlCode = %d.\r\n"),dwIoControlCode));
    switch(dwIoControlCode)
    {
            case IO_ADC_ON:
                    g_adcPara.bOn = TRUE;
                    RETAILMSG(ADC_DBG_MSG,(TEXT("ADC_IOControl: IO_ADC_ON\r\n")));
                    break;
```

```
            case IO_ADC_OFF:
                    g_adcPara.bOn = FALSE;
                    RETAILMSG(ADC_DBG_MSG,(TEXT("ADC_IOControl: IO_ADC_OFF\r\n")));
                    break;
            case IO_ADC_SEL_CHANNEL:
                    if(NULL != lpInBuf)
                    {
                            g_adcPara.nChannel = *lpInBuf;
                            #if 0//用于调试
                            RETAILMSG(ADC_DBG_MSG,  (TEXT("nChannel: 0x%x\r\n"), *lpInBuf));
                            RETAILMSG(ADC_DBG_MSG,  (TEXT("nChannel: 0x%x\r\n"),g_adcPara.nChannel));
                            #endif
                    }
                    else
                    {
                            RETAILMSG(ADC_ERR_MSG,(TEXT("The parameter of  lpInBuf is error \r\n")));
                    }
                    break;
            case IO_ADC_SEL_FREQ:
                    if(NULL != lpInBuf)
                    {
                            dwFreq = (*lpInBuf) * 1000;
                            g_adcPara.nPrescaler = S3C6410_PCLK/dwFreq -1;
                            RETAILMSG(ADC_DBG_MSG,   (TEXT("Prescaler: %d\r\n"),g_
adcPara.nPrescaler));
                    }
                    else
                    {
                            RETAILMSG(ADC_ERR_MSG,(TEXT("The parameter of  Prescaler
is error !\r\n")));
                    }
                    break;
            default:
                    RETAILMSG(ADC_ERR_MSG,(TEXT("ADC_IOControl: default\r\n")));
                    break;
        }

        return TRUE;
    }


    //读ADC转换数据
    DWORD ADC_Read(DWORD hOpenContext,LPVOID pBuffer,DWORD Count)
    {
        int     *pData = NULL;

        if(NULL == pBuffer)
        {
                RETAILMSG(ADC_ERR_MSG, (TEXT("Read buf is null\r\n")));
                return 0;
        }
        pData = pBuffer;
        if(g_adcPara.bOn)
```

```
        {
                *pData = ReadAdc(g_adcPara.nChannel,g_adcPara.nPrescaler);
                return sizeof(int);
        }
        else
        {
                return 0;
        }

        RETAILMSG(ADC_DBG_MSG, (TEXT("ADC_Read---\r\n")));
         return TRUE;
}
```

## 6.4.6 A/D转换程序设计

对ADC驱动的操作和操作LED驱动类似，同属于流驱动。不同之处在于，LED驱动是通过DeviceIoControl()函数发送句柄控制驱动，而ADC驱动则是使用ReadFile()函数读取驱动中的某个值。

1）设计界面。

新建一个名为Adc的MFC项目。首先依次选择“View”→“Resource View”，然后在“Resource View”中依次展开“ADC”→“ADC.rc”→“Dialog”，接着双击“IDD_ADC_DIALOG”，找到工具栏Toolbox，添加Button控件与Edit Control控件，界面如图6-37所示。

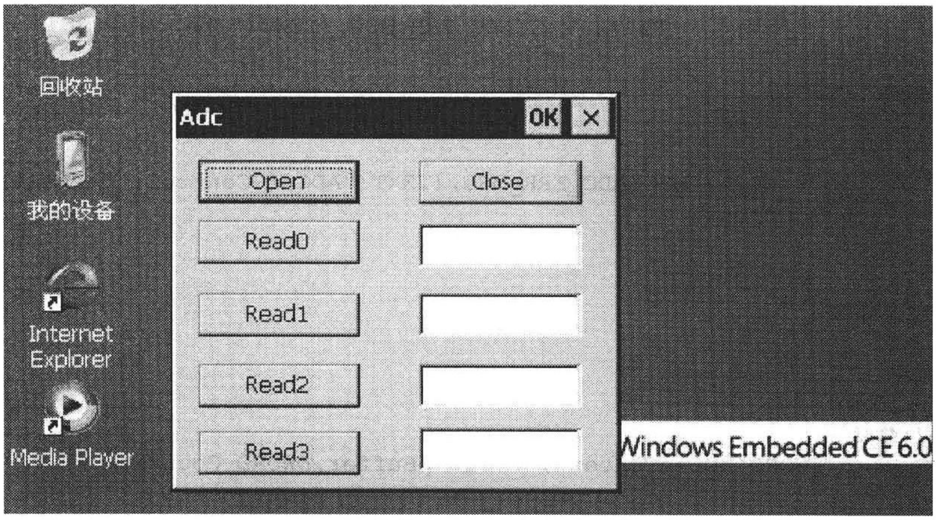



图6-37 ADC软件界面

“Open”按钮和“Close”按钮分别用于打开和关闭驱动。由于该ADC驱动中有4个通道，所以这里使用了4个按钮和4个文本框来分别读取和显示这4个通道中ADC的值。

其中，Button控件的Caption与ID属性分别如表6-26和表6-27所示。

表6-26　Caption属性

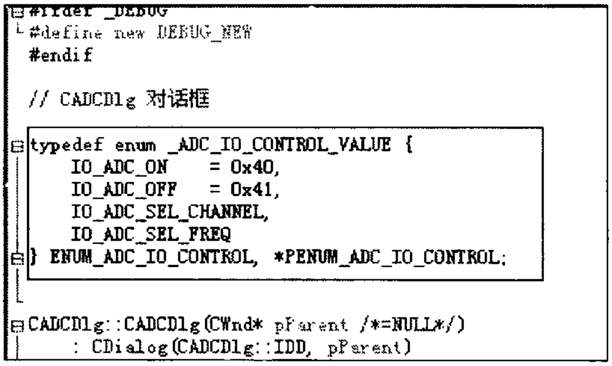
| Caption | ID |
|---|---|
| Open | IDC_BUTTON_OPEN |
| Close | IDC_BUTTON_CLOSE |
| READ0 | IDC_BUTTON_READ0 |
| READ1 | IDC_BUTTON_READ1 |
| READ2 | IDC_BUTTON_READ2 |
| READ3 | IDC_BUTTON_READ3 |

表6-27　ID属性

| ID |
|---|
| IDC_EDIT_READ0 |
| IDC_EDIT_READ1 |
| IDC_EDIT_READ2 |
| IDC_EDIT_READ3 |

2）在ADCDlg.cpp中加入驱动控制程序，如图6-38所示。

在ADCDlg.h中加入public类，如图6-39所示。



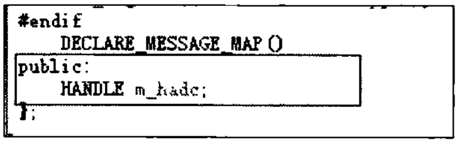
```
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CADCDlg 对话框

typedef enum _ADC_IO_CONTROL_VALUE {
    IO_ADC_ON     = 0x40,
    IO_ADC_OFF    = 0x41,
    IO_ADC_SEL_CHANNEL,
    IO_ADC_SEL_FREQ
} ENUM_ADC_IO_CONTROL, *PENUM_ADC_IO_CONTROL;

CADCDlg::CADCDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CADCDlg::IDD, pParent)
```

图6-38　驱动控制程序



```
#endif
    DECLARE_MESSAGE_MAP()
public:
    HANDLE m_hadc;
};
```

图6-39　在ADCDlg.h中添加public类

3）给“Open”按钮添加程序代码。

在“Open”按钮的选择事件函数中添加以下打开驱动的程序代码。

```
//打开驱动
    m_hadc=CreateFile(_T("ADC1:"),GENERIC_READ|GENERIC_WRITE,0,NULL,OPEN_
EXISTING,0,0);
    //判断打开驱动是否成功
    if(m_hadc == INVALID_HANDLE_VALUE)
    {
            MessageBox(_T("Open ADC Driver error!"));
    }
```

4）给“Close”按钮添加程序代码。

在“Close”按钮的选择事件函数中添加以下关闭驱动的程序代码。

```
if(m_hadc != NULL)
    {
            CloseHandle(m_hadc);
    }
```

5）给“Read0”按钮添加程序代码。

在“Read0”按钮的选择事件函数中添加读取信道0的程序代码，这里使用ReadFile()函数。当在应用中调用该函数时，系统会调用驱动中的XXX_Read()函数。ReadFile()函数的声明如下：

```
BOOL ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

相关参数介绍如下。

- hFile：设备句柄，这里使用m_hadc。
- lpBuffer：存放读取数据的地址。
- nNumberOfBytesToRead：表示要读取的数据长度。
- lpNumberOfBytesRead：表示实际读取了多少数据。
- lpOverlapped：该参数在CE上不支持，设置为NULL。

下面在“Read0”按钮中添加以下程序代码：

```
WORD wData;
   DWORD dwBytes;
   int channel = 0;
   //设置读取通道
   DeviceIoControl(m_hadc,IO_ADC_SEL_CHANNEL, &channel,NULL,NULL,NULL,NULL,NULL);
   //读取数据
   ReadFile(m_hadc, &wData,sizeof(wData), &dwBytes,NULL);
   //把数据显示在Edit控件中
   SetDlgItemInt(IDC_EDIT_READ0,wData);
```

6）给“Read1”按钮添加程序代码。

参照“Read0”按钮的程序代码，修改读取的通道值，然后读取ADC值。

```
WORD wData;
   DWORD dwBytes;
   int channel = 1;
   //设置读取通道
   DeviceIoControl(m_hadc,IO_ADC_SEL_CHANNEL, &channel,NULL,NULL,NULL,NULL,NULL);
   //读取数据
   ReadFile(m_hadc, &wData,sizeof(wData), &dwBytes,NULL);
   //把数据显示在Edit控件中
   SetDlgItemInt(IDC_EDIT_READ1,wData);
```

7）给“Read2”按钮添加程序代码。

同样参照“Read0”按钮的程序代码，修改读取的通道值，然后读取ADC值。

```
WORD wData;
   DWORD dwBytes;
   int channel = 2;
   //设置读取通道
   DeviceIoControl(m_hadc,IO_ADC_SEL_CHANNEL, &channel,NULL,NULL,NULL,NULL,NULL);
   //读取数据
```

```
    ReadFile(m_hadc, &wData,sizeof(wData),  &dwBytes,NULL);
    //把数据显示在Edit控件中
    SetDlgItemInt(IDC_EDIT_READ2,wData);
```

8）给“Read3”按钮添加程序代码。

同样参照“Read0”按钮的程序代码，修改读取的通道值，然后读取ADC值。

```
WORD wData;
    DWORD dwBytes;
    int channel = 3;
    //设置读取通道
    DeviceIoControl(m_hadc,IO_ADC_SEL_CHANNEL, &channel,NULL,NULL,NULL,NULL,NULL);
    //读取数据
    ReadFile(m_hadc, &wData,sizeof(wData),  &dwBytes,NULL);
    //把数据显示在Edit控件中
    SetDlgItemInt(IDC_EDIT_READ3,wData);
```

9）编译程序。

在Visual Studio 2005上方工具栏中间的编译方式中选择“Release”，在SDK下拉菜单中选择“DMA6410（ARMV4I）”。如图6-23所示。

配置好后，依次选择“Build”→“Rebuild ADC”，然后开始编译程序。编译成功后即生成可执行文件ADC.exe，该文件在文件夹“我的文档\Visual Studio 2005\Projects\ADC\ADC\DMA6410（ARMV4I）\Release”中。为了验证ADC转换程序是否正确，需要将ADC.exe复制到开发平台DMA-6410XP中。

10）复制ADC.exe至开发平台验证程序。

打开平台开关，启动Windows Embedded CE 6.0系统。启动后，PC上的ActiveSync软件显示连接成功，双击PC右下角任务栏中的图标，打开ActiveSync软件。

在ActiveSync窗口中单击“浏览”，在PC中找到编译好的ADC.exe文件，将该文件复制到开发平台的NANDFLASH文件夹中，然后操作开发平台，在NANDFLASH文件夹中打开ADC.exe，双击运行程序。

界面打开后，首先单击“Open”按钮开启ADC数/模转换功能。在DMA-6410XP实验平台，有一个温度传感器LM35D，连接到S3C6410的A/D转换器的0通道。单击“Read0”按钮可以读取转换的数字信号量。另外，还有3个模拟信号连接到S3C6410的A/D转换器的1、2、3通道。通过调节3个输入的模拟信号值，分别单击“Read1”、“Read2”和“Read3”按钮，可以读取1、2和3通道转换的数字信号量。

## 6.5 UART串口驱动程序

将PC的串行口与DMA-6410XP实验平台的UART1串口连接。采用Visual Studio 2005编写串行通信程序，在Windows Embedded CE6.0下实现PC与实验平台的UART1串行通信，并将结果显示在LCD屏幕上。

## 6.5.1 串行接口概述

串行通信是微型计算机间常见的近距离通信手段，由于其使用方便、程序编写简单，所以广泛使用。目前，几乎所有的微控制器、PC都提供串行通信接口。

### 1. 串行通信数据传输格式

传送开始前，线路处于空闲状态，送出连续“1”。传送开始时，首先发送一个“0”作为起始位，然后二进制编码数据。每个数据位长可以约定为5位、6位、7位或8位，一般采用ASCII编码，后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验，这样就可以取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续1位、1.5位或2位的时间宽度。

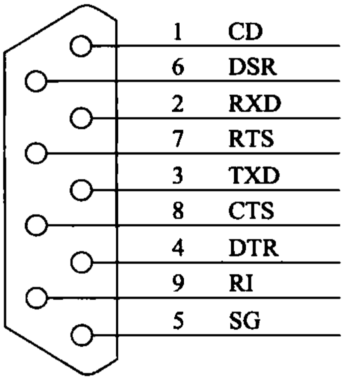



图6-40 DB-9引脚

### 2. 电气特性

RS-232标准采用的接口是9 pin或25pin的D形插头，常用的是9针插头（DB-9），如图6-40所示。

如表6-28所示，要完成基本的通信功能，实际上只需要RXD、TXD、GND即可，但由于RS-232-C标准所定义的高、低电位信号与S3C6410系统的LVTTL电路定义的高、低电位信号完全不同，LVTTL的标准逻辑“1”对应2~3.3V电位，标准逻辑“0”对应0~0.4V电位，而RS-232-C标准采用负逻辑方式，标准逻辑“1”对应-15~-5V电位，标准逻辑“0”对应+5~+15V。显然，两者间要进行通信，必须经过电位的转换，转换芯片采用MAX3243。

表6-28 引脚信号

| 引脚名称 | 英文全称 | 说明 |
|---|---|---|
| RI | Ring Indicator | 响铃侦测 |
| TXD | Transmitted Data | 数据输出线 |
| RXD | Received Data | 数据接收线 |
| RTS | Request to Send | 要求发送资料 |
| CTS | Clear to Send | 响应对方发送的RTS发送请求，告诉对方可以发送 |
| DSR | Data Set Ready | 告知本机在待命状态 |
| DTR | Data Terminal Ready | 告知数据终端处于待命状态 |
| CD | Carrier Detect | 载波检出，用以确认是否收到Modem的载波 |
| SG | Signal Ground | 信号线的接地线 |

### 3. S3C6410的UART控制器

（1）S3C6410的UART接口

S3C6410的UART（通用异步串行端口）单元提供4个独立的异步串行I/O端口，每个都可以在中断和DMA两种模式下运行。它们支持的最高波特率为3Mbit/s。每个UART通道包含两个64位FIFO，分别提供给接收和发送。

S3C6410的UART可以进行以下参数的设置：可编程的波特率、红外收/发模式、1或2个停止位，以及5位、6位、7位或8位数据宽度和奇偶位校验。

每个UART包含波特率发生器、接收器、发送器和控制单元。波特率发生器以MCLK为频率源。发送器和接收器包含16字节FIFO缓存器和移位元缓存器，当发送数据时，数据先写到FIFO，然后复制到发送移位元缓存器，最后从数据输出端口（TxDn）依次被移位输出。接收的数据也同样从接收埠（RxDn）移位输入到移位元缓存器，然后复制到FIFO中。

（2）数据发送操作

数据发送封包的格式是可编译的，它包含一个开始位、5～8个数据位、一个可选的奇偶位和一个或两个停止位，这些可以通过线性控制器（ULCONn）来设置。发送器也能够产生发送中止条件。中止条件件迫使串行端口输出保持在逻辑0状态，这种状态保持一个传输封包的时间长度。通常，在一封包传输数据完整地传输完后，通过这个全0状态将中止信号发送给对方。在中止信号发送后，传送数据连续放到FIFO中（在不使用FIFO模式下，将放到输出保持缓存器中）。

（3）数据接收操作

与数据发送一样，数据接收的封包也是可以编译的，它同样包含一个开始位、5～8个数据位，一个可选的奇偶位和一个或两个停止位，它们都是通过线性控制器位设置的。接收器能够检测溢出错误、奇偶校验错误、封包错误和中止状况，每种情况下都有一个错误标志位。

（4）波特率发生器

每个UART的波特率发生器为传输提供了串行移动位时钟。波特率产生器的时钟可以从S3C6410的内部系统时钟（PCLK）或EXT_UCLK中选择。波特率由时钟（PCLK或UCLK）16分频和UART波特率除数缓存器（UBRDIVn）指定的16位除数决定。UBRDIVn的值可以按照下式确定：

$$UBRDIVn=(int)(PCLK/(bps*16))-1$$

除数的范围为$1\sim2^{16}-1$。例如，如果波特率为115200bit/s且PCLK或UCLK为40MHz，则UBRDIVn为：

$$UBRDIVn =(int)(40000000)(115200*16))-1= (int)(21.7)-1= 21-1 = 20$$

## 6.5.2 UART接口电路

UART接口电路如图6-41所示。

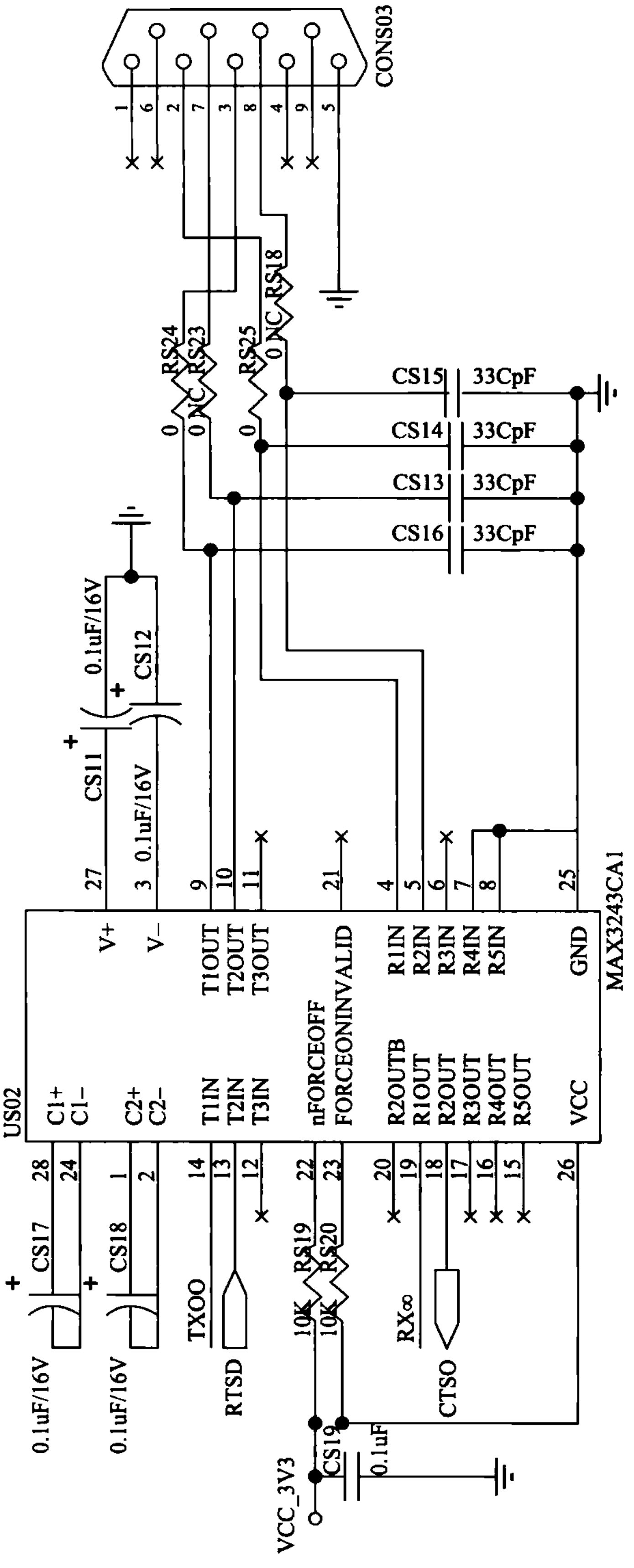



图6-41 UART接口电路

### 6.5.3 UART专用寄存器

下面介绍几个与UART相关的寄存器。

（1）UART行控制寄存器

在UART模块包括4个行控制寄存器，即ULCON0、ULCON1、ULCON2和ULCON3。下面看看行控制寄存器及其位定义，分别如表6-29和表6-30所示。

表6-29 UART行控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| ULCON0 | 0x7F005000 | 读/写 | UART 0通道行控制寄存器 | 0x00 |
| ULCON1 | 0x7F005400 | 读/写 | UART1通道行控制寄存器 | 0x00 |
| ULCON2 | 0x7F005800 | 读/写 | UART2通道行控制寄存器 | 0x00 |
| ULCON3 | 0x7F005C00 | 读/写 | UART3通道行控制寄存器 | 0x00 |

表6-30 UART行控制寄存器的位分配

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Reserved | [7] | 保留 | 0 |
| Infra-Red Mode | [6] | 确定是否采用红外模式：<br>0 = 普通操作模式<br>1= 红外线输出/接收模式 | 0 |
| Parity Mode | [5:3] | 确定奇偶产生类型和校验（在UART发送/接收操作过程中）：<br>0xx = 无校验<br>100 =奇校验<br>101 = 偶校验<br>110 = 奇偶强制/校验为1<br>111 = 奇偶强制/校验为0<br>111 = 强制为0 | 000 |
| Number of Stop Bit | [2] | 确定每帧中停止位个数：<br>0 = 每帧，1位停止位<br>1 = 每帧，2位停止位 | 0 |
| Word Length | [1:0] | 确定每帧中数据位的个数：<br>00 = 5位，01 = 6位<br>10 = 7位，11 = 8位 | 00 |

（2）UART控制寄存器

UART控制寄存器也有4个：UCON0、UCON1、UCON2和UCON3。UART控制寄存器及其位分配分别如表6-31和表6-32所示。

表6-31 UART控制寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| UCON0 | 0x7F005004 | 读/写 | UART 0通道控制器 | 0x00 |
| UCON1 | 0x7F005404 | 读/写 | UART1通道控制器 | 0x00 |
| UCON2 | 0x7F005804 | 读/写 | UART2通道控制器 | 0x00 |
| UCON3 | 0x7F005C04 | 读/写 | UART3通道控制器 | 0x00 |

表6-32 UART控制寄存器的位分配

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Clock Selection | [11:10] | 选择PCLK或者EXT_UCLK0作为UART波特率时钟<br>x0=PCLK:DIV_VAL =（PCLK /（b/s × 16））–1<br>01=EXT_UCLK0:DIV_VAL =（EXT_UCLK /（b/s × 16））–1<br>11=EXT_UCLK1:DIV_VAL=（EXT_UCLK /（b/s × 16））–1 | 0 |
| Tx中断类型 | [9] | 发送中断请求类型：<br>0 = 脉冲 （在非FIFO模式下的发送缓冲区寄存器为空，或者在FIFO模式下发送FIFO达到触发脉冲时产生中断）<br>1 = 电平 （当非FIFO模式下的发送缓冲区寄存器为空，或者在FIFO模式下发送FIFO达到触发电平时产生中断） | 0 |
| Rx中断类型 | [8] | 接收中断请求类型：<br>0 = 脉冲（在非FIFO模式下接收缓冲区寄存器接收数据，或者在FIFO模式下发送FIFO达到触发脉冲时产生中断）<br>1 = 电平 （在非FIFO模式下接收缓冲区寄存器接收数据，或者在FIFO模式下发送FIFO达到触发电平时产生中断） | 0 |
| Rx Time Out Enable | [7] | 使能/禁止接收超时中断，当UART FIFO使能时：<br>0 = 禁止；1 = 使能 | 0 |
| Rx Error Status Interrupt Enable | [6] | 在接收过程中，如果发生帧错误或溢出错误，使能/禁止UART产生中断<br>0 = 不产生接收错误状态中断<br>1 = 产生接收错误状态中断 | 0 |
| Loop-back Mode | [5] | 设置环回位为1，使UART进入环回模式。这种模式仅为测试目的使用<br>0 = 普通操作；1 = 环回模式 | 0 |
| Send Break Signal | [4] | 置环回位为1使UART进入环回模式。这种模式只为测试提供参考<br>0=正常发送；1= 发送中断信号 | 0 |
| Transmit Mode | [3:2] | 确定哪个模式可以将数据写到UART发送缓冲寄存器<br>00= 禁止<br>01= 中断请求或轮询模式<br>10= DMA0请求（仅用于UART0），DMA3请求 （请求信号0）<br>11= DMA1请求（仅请求信号1） | 00 |

（续）

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Receive Mode | [1:0] | 确定哪个模式可以从UART接收缓冲寄存器读数据<br>00 =禁止<br>01 = 中断请求或轮询模式<br>10= DMA0请求（仅用于UART0），DMA3请求（请求信号0）<br>11= DMA1请求（请求信号1） | 00 |

（3）UART接收（Rx）/发送（Tx）状态寄存器

UART模块有4个UART接收/发送状态寄存器：UTRSTAT0、UTRSTAT1、UTRSTAT2和UTRSTAT3。UART接收/发送状态寄存器及其位分配分别如表6-33和表6-34所示。

表6-33 UART接收/发送状态寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| UTRSTAT0 | 0x7F005010 | 读 | UART 0通道接收/发送状态寄存器 | 0x6 |
| UTRSTAT1 | 0x7F005410 | 读 | UART 1通道接收/发送状态寄存器 | 0x6 |
| UTRSTAT2 | 0x7F005810 | 读 | UART 2通道接收/发送状态寄存器 | 0x6 |
| UTRSTAT3 | 0x7F005C10 | 读 | UART 3通道接收/发送状态寄存器 | 0x6 |

表6-34 UART接收/发送状态寄存器的位分配

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| Transmitter Empty | [2] | 在发送缓冲寄存器没有有效数据或发送移位寄存器为空时，该位自动置1<br>0 = 不空<br>1 = 发送器（发送缓冲寄存器和移位寄存器）空 | 1 |
| Transmit Buffer Empty | [1] | 当发送缓冲寄存器为空时，该位自动置1<br>0 =发送缓冲寄存器不空<br>1 =空<br>（在非FIFO模式，中断或DMA被申请。在FIFO模式，当Tx FIFO触发水平设置为00（空）时，中断或DMA被申请。）<br>如果UART使用FIFO，则用户应该检查UFSTAT寄存器的Tx FIFO计数位和Tx FIFO满标志位，以代替检查该位 | 1 |
| Receive Data Ready Buffer | [0] | 无论何时接收缓冲寄存器包含在RXDn接口接收的有效数据，该位自动置1<br>0 = 空<br>1 = 接收缓冲寄存器有接收数据（在非FIFO模式，中断或DMA被申请）<br>如果UART使用FIFO，则用户应该检查UFSTAT寄存器中的Rx FIFO计数位和Rx FIFO满标志位以代替检查该位 | 0 |

（4）UART发送缓冲寄存器（保存寄存器和FIFO寄存器）

UART模块有4个发送缓冲寄存器：UTXH0、UTXH1、UTXH2和UTXH3，如表6-35所示。UTXHn有一个8位数据作为发送数据。

表6-35 UART发送缓冲寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| UTXH0 | 0x7F005020 | 写 | UART 0通道发送缓冲寄存器 | |
| UTXH1 | 0x7F005420 | 写 | UART 1通道发送缓冲寄存器 | |
| UTXH2 | 0x7F005820 | 写 | UART 2通道发送缓冲寄存器 | |
| UTXH3 | 0x7F005C20 | 写 | UART 3通道发送缓冲寄存器 | |

（5）UART接收缓冲寄存器（保存寄存器和FIFO寄存器）

UART模块有4个接收缓冲寄存器：URXH0、URXH1、URXH2和URXH3，如表6-36所示。URXHn有一个8位数据作为发送数据。

表6-36 UART接收缓冲寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
|---|---|---|---|---|
| URXH0 | 0x7F005024 | 读 | UART 0通道接收缓冲寄存器 | |
| URXH1 | 0x7F005424 | 读 | UART 1通道接收缓冲寄存器 | |
| URXH2 | 0x7F005824 | 读 | UART 2通道接收缓冲寄存器 | |
| URXH3 | 0x7F005C24 | 读 | UART 3通道接收缓冲寄存器 | |

（6）UART波特率分频寄存器

UART模块有4个波特率分频寄存器：UBRDIV0、UBRDIV1、UBRDIV2和UBRDIV3。UART波特率分频寄存器及其位分配分别如表6-37和表6-38所示。

UBRDIVn中的值决定串行Tx/Rx时钟波特率，如下：

$$\text{DIV_VAL} = \text{UBRDIVn} + (\text{UDIVSLOTn中1的量})/16$$

$$\text{DIV_VAL} = (\text{PCLK}/(\text{b/s}\times 16))-1$$

$$\text{DIV_VAL} = (\text{EXT_UCLK0}/(\text{b/s}\times 16).)-1$$

或者

$$\text{DIV_VAL} = (\text{EXT_UCLK1}/(\text{b/s}\times 16))-1$$

除数的范围为1～$2^{16}-1$，并且UEXTCLK应该比PCLK小。

利用UDIVSLOT能够得到更准确的波特率。例如，如果波特率是115 200 bit/s，PCLK、EXT_UCLK0或EXT_UCLK1是40 MHz则UBRDIVn和UDIVSLOTn计算结果如下：

$$\text{DIV_VAL} = (40000000/(115200\times 16))-1 = 21.7-1 = 20.7$$

UBRDIVn = 20（DIV_VAL的整数部分）

（UDIVSLOTn中1的数量）/16 = 0.7

这时，（UDIVSLOTn中1的数量）= 11。

表6-37 UART波特率分频寄存器

| 寄 存 器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| UBRDIV0 | 0x7F005028 | 读 | 波特率分频寄存器0 | 0x0000 |
| UBRDIV 1 | 0x7F005428 | 读 | 波特率分频寄存器1 | 0x0000 |
| UBRDIV 2 | 0x7F005824 | 读 | 波特率分频寄存器2 | 0x0000 |
| UBRDIV 3 | 0x7F005C28 | 读 | 波特率分频寄存器3 | 0x0000 |

表6-38 波特率分频寄存器的位分配

| 位 名 称 | 位 | 描 述 | 初始状态 |
|---|---|---|---|
| UBRDIVn | [15:0] | 波特率分频寄存器的值（当UART时钟是PCLK时，UBRDIVn必须大于0） | |

## 6.5.4 串行通信驱动程序分析

### 1. UART驱动的程序结构

UART有关程序代码在BSP中的SRC\Drivers\SERIAL目录下。可以在Windows CE的%OSROOT%\PUBLIC\COMMON\OAK\DRIVERS\SERIAL目录中找到微软公司提供的UART驱动原始程序代码（注意：%OSROOT%是指Windows CE的安装目录）。这些程序代码的作用是把UART通用的特性整合起来，以达到程序代码和数据的多功能。每个串行端口UART的缓存器都有一份副本，不同之处仅在于它们的物理地址，方便使用者容易地使用这个功能。类层次图如图6-42所示。

CRegisterEdit：对注册表的读/写。

CSerialPDD：继承CRegisterEdit是为了方便注册表的读/写，因为驱动的很多配置信息是存储于注册表中的。

CPdd6410Uart：实现对串行埠的通用操作，CReg6410Uart对象作为它的成员，方便对缓存器读/写。

CPdd6410Serial0、CPdd6410Serial1、CPdd6410Serial2、CPdd6410Serial3：这4个类继承自CPdd6410Uart，主要实现对于串行端口不同特性的配置，如缓存器的位地址配置为IO接口功能。总之，它实现与标准不符的部分或参数的调整，其中最为核心的程序代码在CPdd6410Uart中。

### 2. 函数库的调用分析

在BSP的SRC\Drivers\SERIAL目录下，有一个Source文件，包含编译需要的库名及路径，如图6-43所示。

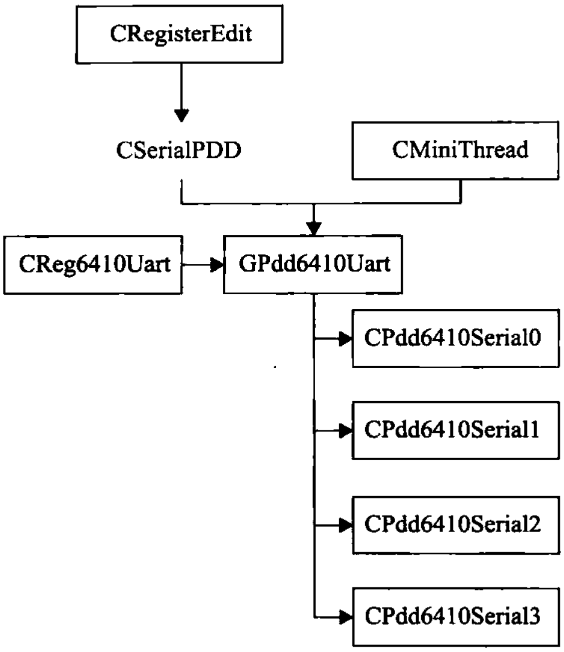



图6-42 类层次图

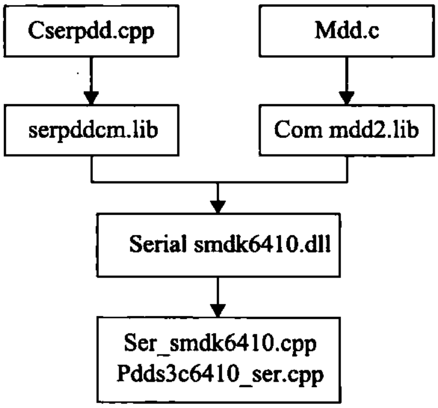



图6-43 编译需要的库名及路径

## 6.5.5 串行通信底层驱动程序设计

```
#include <windows.h>
#include <types.h>
#include <ceddk.h>
#include <bsp.h>
#include <ddkreg.h>
#include <serhw.h>
#include <Serdbg.h>
#include <pdds3c6410_ser.h>
#include <s3c6410_base_regs.h>

#define    EPLL_CLK    0

CReg6410Uart::CReg6410Uart(PULONG pRegAddr)
{
    m_fIsBackedUp = FALSE;
    m_ULCONBackup = NULL;
    m_UCONBackup = NULL;
    m_UFCONBackup = NULL;
    m_UMCOMBackup = NULL;
    m_UBRDIVBackup = NULL;
    m_UDIVSLOTBackup = NULL;
    m_UINTMBackup = NULL;

    m_BaudRate = NULL;
    m_s3c6410_pclk = S3C6410_PCLK;
}
```

```
BOOL CReg6410Uart::Init()
{
    BOOL bRet = TRUE;

    if(m_pReg)
    { // 设置值为默认
        Write_ULCON(0);
        Write_UCON(0);
        Write_UFCON(0);
        Write_UMCON(0);
        bRet = TRUE;
    }
    else
    {
        bRet = FALSE;
    }

    return bRet;
}


void CReg6410Uart::Backup()
{
    m_ULCONBackup = Read_ULCON();
    m_UCONBackup = Read_UCON();
    m_UFCONBackup = Read_UFCON();
    m_UMCOMBackup = Read_UMCON();
    m_UBRDIVBackup = Read_UBRDIV();
    m_UDIVSLOTBackup = Read_UDIVSLOT();
    m_UINTMBackup = Read_UINTM();
    m_fIsBackedUp = TRUE;
}

void CReg6410Uart::Restore()
{
    if(m_fIsBackedUp)
    {
        Write_ULCON(m_ULCONBackup);
        Write_UCON(m_UCONBackup);
        Write_UFCON(m_UFCONBackup);
        Write_UMCON(m_UMCONBackup);
        Write_UBRDIV(m_UBRDIVBackup);
        Write_UDIVSLOT(m_UDIVSLOTBackup);
        Write_UINTM(m_UINTMBackup);
        m_fIsBackedUp = FALSE;
    }
}

CReg6410Uart::Write_BaudRate(ULONG BaudRate)
{
    DOUBLE Div_val;
    UINT UDIVSLOTn = 0;
```

```
    UINT UBRDIV = 0;
    BOOL bRet = TRUE;

    DEBUGCHK(BaudRate!= NULL);

    DEBUGMSG(ZONE_INIT, (TEXT("SetBaudRate -> %d\r\n"),BaudRate));

    if( (Read_UCON() & UART_CS_MASK) == UART_CS_PCLK)
    {
        Div_val = (m_s3c6410_pclk/16.0/BaudRate);
        UBRDIV = (int)Div_val - 1;
        Write_UBRDIV(UBRDIV);
        UDIVSLOTn = (int)( (Div_val - (int)Div_val) * 16);
        Write_UDIVSLOT(UDIVSLOT_TABLE[UDIVSLOTn]);
           RETAILMSG(FALSE,  (TEXT("CLK:%d,BaudRate:%d,UBRDIV:%d,UDIVSLOTn:%d\r\
n"),m_s3c6410_pclk,BaudRate,UBRDIV,UDIVSLOTn));
        bRet = TRUE;
    }
    else if( (Read_UCON() & UART_CS_MASK) == UART_CS_EPLLCLK)
    {
        Div_val = (S3C6410_ECLK/16.0/BaudRate);
        UBRDIV = (int)Div_val - 1;
        Write_UBRDIV(UBRDIV);
        UDIVSLOTn = (int)( (Div_val - (int)Div_val) * 16);
        Write_UDIVSLOT(UDIVSLOT_TABLE[UDIVSLOTn]);
           RETAILMSG(FALSE,  (TEXT("CLK:%d,BaudRate:%d,UBRDIV:%d,UDIVSLOTn:%d\r\
n"),S3C6410_ECLK,BaudRate,UBRDIV,UDIVSLOTn));
        bRet = TRUE;
    }
    else
    {
         RETAILMSG(TRUE, (TEXT("ERROR: The s3c6410a serial driver doesn't support
an external UART clock.\r\n")));
        ASSERT(FALSE);
        bRet =FALSE;
    }

    return bRet;
}
#ifdef DEBUG

void CPdd6410Uart::PostInit()
{
    DWORD dwCount=0;
    m_HardwareLock.Lock();
    m_pReg6410Uart->Write_UCON(0); // 设置为默认
     DisableInterrupt(S6410UART_INT_RXD | S6410UART_INT_TXD | S6410UART_INT_ERR |
S6410UART_INT_MODEM);
    // 屏蔽所有的中断
      while((GetInterruptStatus() & (S6410UART_INT_RXD | S6410UART_INT_TXD |
S6410UART_INT_ERR | S6410UART_INT_MODEM))!=0 && dwCount <MAX_RETRY)
    {
```

```
            InitReceive(TRUE);
            InitLine(TRUE);
             ClearInterrupt(S6410UART_INT_RXD | S6410UART_INT_TXD | S6410UART_INT_ERR
| S6410UART_INT_MODEM);
            dwCount++;
        }
          ASSERT((GetInterruptStatus() & (S6410UART_INT_RXD | S6410UART_INT_TXD |
S6410UART_INT_ERR | S6410UART_INT_MODEM))==0);
        m_HardwareLock.Unlock();
        CSerialPDD::PostInit();
        CeSetPriority(m_dwPriority256);
    #ifdef DEBUG
        if(ZONE_INIT)
        {
            m_pReg6410Uart->DumpRegister();
        }
    #endif
        ThreadStart();  // 开始IST
    }
    DWORD CPdd6410Uart::ThreadRun()
    {
        DWORD dwData;
        DWORD interrupts=NULL;

        while(m_hISTEvent!=NULL && !IsTerminated() )
        {
            if(WaitForSingleObject(m_hISTEvent,m_dwISTTimeout) == WAIT_OBJECT_0)
            {
                m_HardwareLock.Lock();

                while( !IsTerminated() )
                {
                     dwData = (GetInterruptStatus() & (S6410UART_INT_RXD | S6410UART_
INT_TXD | S6410UART_INT_ERR | S6410UART_INT_MODEM) );
                    if(dwData)
                    {
                         DEBUGMSG(ZONE_THREAD, (TEXT("CPdd6410Uart::ThreadRun Active
INT=%x\r\n"),dwData));

                        if((dwData & S6410UART_INT_RXD)!=0)
                        {
                            interrupts |= INTR_RX;
                        }
                        if((dwData & S6410UART_INT_TXD)!=0)
                        {
                            interrupts |= INTR_TX;
                        }
                        if((dwData & S6410UART_INT_ERR)!=0)
                        {
                            interrupts |= INTR_LINE | INTR_RX;
                        }
                        if((dwData & S6410UART_INT_MODEM)!=0)
```

```
                    {
                        interrupts |=INTR_MODEM;
                    }

                    NotifyPDDInterrupt( (INTERRUPT_TYPE)interrupts);

                    ClearInterrupt(dwData);
                }
                else
                {
                    break;
                }
            }

            m_HardwareLock.Unlock();

            InterruptDone(m_dwSysIntr);
        }
        else
        {
                DEBUGMSG(ZONE_THREAD,(TEXT("CPdd6410Uart::ThreadRun timeout
INT=%x,MASK=%d\r\n"),m_pReg6410Uart->Read_UINTP(),m_pReg6410Uart->Read_UINTM()) );
            ASSERT(FALSE);
        }
    }

    return 1;
}
BOOL CPdd6410Uart::InitialEnableInterrupt(BOOL bEnable)
{
    m_HardwareLock.Lock();
    if(bEnable)
    {
        EnableInterrupt(S6410UART_INT_RXD | S6410UART_INT_ERR | S6410UART_INT_MODEM);
    }
    else
    {
        DisableInterrupt(S6410UART_INT_RXD | S6410UART_INT_ERR | S6410UART_INT_MODEM);
    }
    m_HardwareLock.Unlock();
    return TRUE;
}

BOOL  CPdd6410Uart::InitXmit(BOOL bInit)
{
    DWORD dwTicks = 0;
    DWORD dwUTRState;

    if(bInit)
    {
        m_HardwareLock.Lock();
        DWORD dwBit = m_pReg6410Uart->Read_UCON();
```

```
    #if EPLL_CLK
            // 设置UART时钟
            dwBit &= ~(UART_CS_MASK);
            dwBit |= (UART_CS_EPLLCLK);
    #endif
            // 设置发送中断请求类型,设置发送中断或轮询模式
            dwBit &= ~(UART_TX_INT_TYPE_MASK|UART_TX_MODE_MASK);
            dwBit |= (UART_TX_INT_TYPE_LEVEL|UART_TX_INT_POLL);
            m_pReg6410Uart->Write_UCON(dwBit);

            dwBit = m_pReg6410Uart->Read_UFCON();
            // 复位发送FIFO寄存器
            dwBit |= (UART_FCR_TX_FIFO_RESET);
            dwBit &= ~(UART_FCR_FIFO_ENABLE);
            m_pReg6410Uart->Write_UFCON(dwBit);
            // 设置触发电平为16
            dwBit &= ~(UART_FCR_TX_FIFO_TRIG_MASK);
            dwBit |= (UART_FCR_TX_FIFO_TRIG_16);    //16
            m_pReg6410Uart->Write_UFCON(dwBit);
            // 发送FIFO寄存器允许
            dwBit &= ~(UART_FCR_TX_FIFO_RESET);
            dwBit |= (UART_FCR_FIFO_ENABLE);
            m_pReg6410Uart->Write_UFCON(dwBit); // 发送FIFO寄存器复位完成
            m_HardwareLock.Unlock();
        }
        else
        { // 确保数据被传送输出
            // 必须确保传输完成,因为MDD将关闭该设备后返回
             while(dwTicks < TIMEOUT_TX_EMPTY && ((dwUTRState = m_pReg6410Uart->Read_
UTRSTAT()) & (UART_TRANSMITTER_STATUS_MASK | UART_TX_BUFFER_STATUS_MASK))
                != (UART_TRANSMITTER_EMPTY | UART_TX_BUFFER_EMPTY))
            { // 发送缓冲寄存器不为空
                 DEBUGMSG(ZONE_THREAD|ZONE_WRITE, (TEXT("CPdd6410Uart::InitXmit! Wait
for UTRSTAT=%x clear.\r\n"),dwUTRState));
                Sleep(5);
                dwTicks +=5;
            }
        }
        return TRUE;
    }
    DWORD   CPdd6410Uart::GetWriteableSize()
    {
        DWORD dwWriteSize = 0;
        DWORD dwUfState = m_pReg6410Uart->Read_UFSTAT() ;
        if((dwUfState & UART_FSR_TX_FIFO_MASK)==UART_FSR_TX_FIFO_NOT_FULL)
        {
              dwUfState = ((dwUfState & UART_FSR_TX_FIFO_COUNT_MASK) >> UART_FSR_TX_
FIFO_COUNT_SHFIT); // 这是FIFO寄存器计数
            if(dwUfState < S3C6410_UART_FIFO_DEPTH-1)
            {
                dwWriteSize = S3C6410_UART_FIFO_DEPTH-1 - dwUfState;
            }
```

```
    }
    return dwWriteSize;
}

void CPdd6410Uart::XmitComChar(UCHAR ComChar)
{
    // 此函数具有轮询,直到数据可以发送出去
    BOOL bDone = FALSE;
    do
    {
        m_HardwareLock.Lock();
        if(GetWriteableSize()!=0)
        {
            m_pReg6410Uart->Write_UTXH(ComChar);
            bDone = TRUE;
        }
        else
        {
            EnableXmitInterrupt(TRUE);
        }
            m_HardwareLock.Unlock();
        if(!bDone)
        {
            WaitForSingleObject(m_XmitFlushDone, (ULONG)1000);
        }
    }while(!bDone);
}
BOOL    CPdd6410Uart::EnableXmitInterrupt(BOOL fEnable)
{
    m_HardwareLock.Lock();
    if(fEnable)
    {
        EnableInterrupt(S6410UART_INT_TXD);
    }
    else
    {
        DisableInterrupt(S6410UART_INT_TXD);
    }
    m_HardwareLock.Unlock();
    return TRUE;
}
BOOL CPdd6410Uart::CancelXmit()
{
    return InitXmit(TRUE);
}
static PAIRS s_HighWaterPairs[] =
{
    {0,1 },
    {1,8 },
    {2,16 },
    {3,32 }
};
```

```
BYTE  CPdd6410Uart::GetWaterMarkBit()
{
    BYTE bReturnKey = (BYTE)s_HighWaterPairs[0].Key;
    for(DWORD dwIndex = dim(s_HighWaterPairs)-1 ; dwIndex != 0; dwIndex --)
    {
        if(m_dwWaterMark >= s_HighWaterPairs[dwIndex].AssociatedValue)
        {
            bReturnKey = (BYTE)s_HighWaterPairs[dwIndex].Key;
            break;
        }
    }
    return bReturnKey;
}
DWORD CPdd6410Uart::GetWaterMark()
{
    BYTE bReturnValue = (BYTE)s_HighWaterPairs[0].AssociatedValue;
    for(DWORD dwIndex = dim(s_HighWaterPairs)-1 ; dwIndex != 0; dwIndex--)
    {
        if(m_dwWaterMark >= s_HighWaterPairs[dwIndex].AssociatedValue)
        {
            bReturnValue = (BYTE)s_HighWaterPairs[dwIndex].AssociatedValue;
            break;
        }
    }
    return bReturnValue;
}

// 接收
BOOL CPdd6410Uart::InitReceive(BOOL bInit)
{
    m_HardwareLock.Lock();
    if(bInit)
    {
        BYTE uWarterMarkBit = GetWaterMarkBit();
        if(uWarterMarkBit> 3)
        {
            uWarterMarkBit = 3;
        }
        // 设置接收FIFO寄存器
        // 复位接收FIFO寄存器
        DWORD dwBit = m_pReg6410Uart->Read_UFCON();
        dwBit |= (UART_FCR_RX_FIFO_RESET);
        dwBit &= ~(UART_FCR_FIFO_ENABLE);
        m_pReg6410Uart->Write_UFCON(dwBit);
        // 设置触发电平位
        dwBit &= ~(UART_FCR_RX_FIFO_TRIG_MASK);
        dwBit |= (uWarterMarkBit<<4);
        m_pReg6410Uart->Write_UFCON(dwBit);
        // 允许接收FIFO寄存器
        dwBit &= ~(UART_FCR_RX_FIFO_RESET);
        dwBit |= (UART_FCR_FIFO_ENABLE);
```

```
            m_pReg6410Uart->Write_UFCON(dwBit); // 接收FIFO寄存器复位完成
            m_pReg6410Uart->Read_UERSTAT(); // 清除中断
            dwBit = m_pReg6410Uart->Read_UCON();
    #if EPLL_CLK
            // 设置UART时钟
            dwBit &= ~(UART_CS_MASK);
            dwBit |= (UART_CS_EPLLCLK);
    #endif
            // 设置RX中断请求类型,接收超时,RX中断/轮询模式
            dwBit &= ~(UART_RX_INT_TYPE_MASK|UART_RX_TIMEOUT_MASK|UART_RX_MODE_MASK);
            dwBit |= (UART_RX_INT_TYPE_LEVEL|UART_RX_TIMEOUT_ENABLE|UART_RX_INT_POLL);
            m_pReg6410Uart->Write_UCON(dwBit);
            EnableInterrupt(S6410UART_INT_RXD | S6410UART_INT_ERR);
        }
        else
        {
            DisableInterrupt(S6410UART_INT_RXD | S6410UART_INT_ERR);
        }
        m_HardwareLock.Unlock();
        return TRUE;
    }
    ULONG CPdd6410Uart::ReceiveInterruptHandler(PUCHAR pRxBuffer,ULONG *pBufflen)
    {
         DEBUGMSG(ZONE_THREAD|ZONE_READ,(TEXT("+CPdd6410Uart::ReceiveInterruptHandler
pRxBuffer=%x,*pBufflen=%x\r\n"),
            pRxBuffer,pBufflen!=NULL?*pBufflen:0));
        DWORD dwBytesDropped = 0;
        if(pRxBuffer && pBufflen)
        {
            DWORD dwBytesStored = 0 ;
            DWORD dwRoomLeft = *pBufflen;
            m_bReceivedCanceled = FALSE;
            m_HardwareLock.Lock();

            while(dwRoomLeft && !m_bReceivedCanceled)
            {
                ULONG ulUFSTATE = m_pReg6410Uart->Read_UFSTAT();
                DWORD dwNumRxInFifo = ((ulUFSTATE & UART_FSR_RX_FIFO_COUNT_MASK) >> UART_FSR_
RX_FIFO_COUNT_SHIFT);
                if((ulUFSTATE & (UART_FSR_RX_FIFO_MASK))== UART_FSR_RX_FIFO_FULL) // 溢出
                {
                    dwNumRxInFifo = S3C6410_UART_FIFO_DEPTH;
                }
                 DEBUGMSG(ZONE_THREAD|ZONE_READ,(TEXT("CPdd6410Uart::ReceiveInterruptH
andler ulUFSTATE=%x,UTRSTAT=%x,dwNumRxInFifo=%X\r\n"),
                    ulUFSTATE,m_pReg6410Uart->Read_UTRSTAT(),dwNumRxInFifo));
                if(dwNumRxInFifo)
                {
                    ASSERT((m_pReg6410Uart->Read_UTRSTAT() & UART_RX_BUFFER_STATUS_MASK)!=
UART_ RX_BUFFER_EMPTY);
                    while(dwNumRxInFifo && dwRoomLeft)
                    {
```

```
                        UCHAR uLineStatus = GetLineStatus();
                        UCHAR uData = m_pReg6410Uart->Read_URXH();
                        if(DataReplaced(&uData,(uLineStatus & UART_LSR_PE)!=0))
                        {
                            *pRxBuffer++ = uData;
                            dwRoomLeft--;
                            dwBytesStored++;
                        }
                        dwNumRxInFifo --;
                    }
                }
                else
                {
                    break;
                }
            }
            if(m_bReceivedCanceled)
            {
                dwBytesStored = 0;
            }

            m_HardwareLock.Unlock();
            *pBufflen = dwBytesStored;
        }
        else
        {
            ASSERT(FALSE);
        }
         DEBUGMSG(ZONE_THREAD|ZONE_READ,(TEXT("-CPdd6410Uart::ReceiveInterruptHandler
pRxBuffer=%x,*pBufflen=%x,dwBytesDropped=%x\r\n"),
            pRxBuffer,pBufflen!=NULL?*pBufflen:0,dwBytesDropped));
        return dwBytesDropped;
    }
    ULONG CPdd6410Uart::CancelReceive()
    {
        m_bReceivedCanceled = TRUE;
        m_HardwareLock.Lock();
        InitReceive(TRUE);
        m_HardwareLock.Unlock();
        return 0;
    }
    BOOL CPdd6410Uart::InitModem(BOOL bInit)
    {
        m_HardwareLock.Lock();
        m_pReg6410Uart->Write_UMCON(m_pReg6410Uart->Read_UMCON()
            |UART_MCR_AUTO_FLOW_DISABLE
            |UART_MCR_MODEM_INTERRUPT_ENABLE
            |UART_MCR_SET_RTS);
        m_HardwareLock.Unlock();
        return TRUE;
    }
```

```
ULONG CPdd6410Uart::GetModemStatus()
{
    m_HardwareLock.Lock();
    ULONG ulReturn =0 ;
    ULONG Events = 0;
    UINT8 ubModemStatus = (UINT8)m_pReg6410Uart->Read_UMSTAT();
    m_HardwareLock.Unlock();

    // 事件通知
    if(ubModemStatus & (UART_MSR_DCTS))
    {
        Events |= EV_CTS;
    }
    if(Events!=0)
    {
        EventCallback(Events);
    }

    // 查看调制解调器状态
    if(ubModemStatus & (UART_MSR_CTS))
    {
        ulReturn |= MS_CTS_ON;
    }
    return ulReturn;
}
```

### 6.5.6 串口通信程序设计

所谓的串行端口通信程序，其实就是对串行口的读/写操作。在嵌入式的设备上，使用串行端口进行通信的时候很多，掌握对串行口的读/写操作是很有必要的。

首先，创建一个名为Thread的MFC项目，在该窗体上添加一个按钮，设置它的ID为IDC_BUTTON_AAA，Caption为AAA，并在按钮点选事件的函数中添加以下程序代码：

```
while(1)
{
        RETAILMSG(1, (TEXT("AAAAAAAAAA\r\n")));
}
```

这是一个无穷循环，在不断地打印“AAAAAAAAAA”字符串。使用串行口线，把PC串行口和实验平台的串行口连接，然后开启DNW工具，在编译执行的软件后，单击“AAA”按钮，将在DNW工具上看到很多的“AAAAAAAAAA”字符串，并且一直在打印。

现在增加一个功能，在打印“AAAAAAAAAA”的同时还要打印“BBBBBBBBBB”，该怎么做呢?

可以把上面的程序代码改写成如下形式：

```
while(1)
{
        RETAILMSG(1, (TEXT("AAAAAAAAAA\r\n")));
        RETAILMSG(1, (TEXT("BBBBBBBBBB\r\n")));
}
```

现在再增加一个要求，不能修改while循环中的程序代码。那么，该怎么实现呢？可以添加一个称作“BBB”的按钮，并在里面添加以下的程序代码：

```
while(1)
   {
          RETAILMSG(1, (TEXT("BBBBBBBBBB\r\n")));
   }
```

使用一个线程来打印这个字符串。

首先，在OnBnClickedButtonAaa函数中添加如下的程序代码：

```
//线程函数
UINT Thread(LPVOID pParam)
{
          while(1)
          {
                 RETAILMSG(1, (TEXT("BBBBBBBBBB\r\n")));
          }
}
```

然后，在OnBnClickedButtonAaa函数的while循环中添加如下程序代码：

```
DWORD m_dwThreadID;
//启动线程
   HANDLE m_hThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)Thread,this,0,
&m_dwThreadID);
```

现在，重新编译并执行程序，在DNW中将看到有两种字符串的打印信息。

使用线程时，只需要把想要“同时”执行的一段程序代码放到一个函数中，然后使用CreateThread函数将它“执行”。了解了线程的相关知识，下面介绍如何编写串行通信软件程序。

（1）综合分析

驱动串行端口通信是对指定串行口的读/写操作，串行口读操作是使用一个线程不断地检测和读取串行端口中接收的信息，并把该信息显示在文本框中。串行口写操作是把某个文本框中的信息通过WriteFile函数写到串行口，让它发送出去。

（2）建立UART工程

首先，打开计算机桌面上的Visual Studio 2005，依次选择“File”→“New”→“Project”，建立名为Uart的MFC程序，并添加Button控件与Edit Control控件，如图6-44所示。

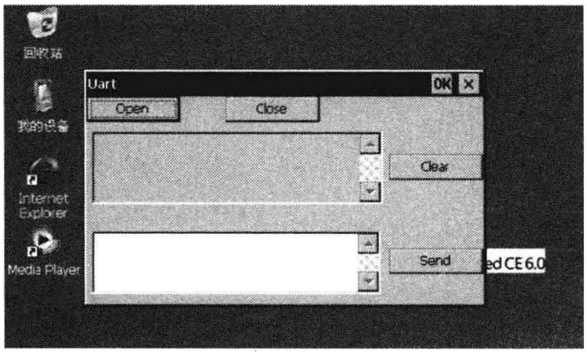


图6-44 UART软件界面

设置“Open”按钮的ID为“IDC_BUTTON_OPEN”，“Caption”为“Open”，该按钮用来打开指定的串行口。设置“Close”按钮的ID为“IDC_BUTTON_CLOSE”，“Caption”为“Close”，该按钮用来关闭打开的串行口。设置“Clear”按钮的ID为“IDC_BUTTON_CLEAR”，“Caption”为“Clear”。设置“Send”按钮的ID为“IDC_ BUTTON_SEND”，“Caption”为“Send”。设置

图6-52中上面的文本框ID为“IDC_EDIT_ RECEIVE”，“Read Only”为“True”，“Multiline”为“True”，“Auto HScroll”为“False”，“Auto VScroll”为“True”，“Vertical Scroll”为“True”。设置图6-52中下面的文本框ID为“IDC_EDIT_ SEND”，“Multiline”为“True，Auto HScroll”为“False”，“Auto VScroll”为“True”，“Vertical Scroll”为“True”。

（3）编写线程函数ReadThread

把下列程序代码加入到UartDlg.cpp中。

```
//接收串行端口数据线程
UINT ReadThread(LPVOID pParam)
{
   CUartDlg *Pthis = (CUartDlg*)pParam ; /*转换参数为CUartDlg类型,便于以后使用它的成员变
量和函数等*/
   if(Pthis == NULL)      //检查参数是否有效
          return 0;
   BYTE   byData;
   DWORD   dwBytes;
   BYTE    byReceiveBuf[1024];  //缓冲区的大小
   int     nBytes= 0;
   DWORD  evtMask;

   //设置字符监听信号
   SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
   //只要m_hcom有效,就一直循环
   while(Pthis->m_hcom != NULL && Pthis->m_hcom != INVALID_HANDLE_VALUE)
   {
          //等待收到串行端口信号
          if(WaitCommEvent(Pthis->m_hcom, &evtMask,0))
          {
                 //设置字符监听信号
                 SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
                 //清空接收数组
                 memset(byReceiveBuf,0,1024);
                 nBytes = 0;
                 do
                 {
                        //读字符
                        if(ReadFile(Pthis->m_hcom, &byData,1, &dwBytes,0))
                        {
                               if(dwBytes == 1)
                               {
                                      byReceiveBuf[nBytes++] = byData;
                                      if(nBytes == 1024)
                                      {
                                             break;  //接收缓冲区已满
                                      }
                               }
                        }
                 } while(dwBytes == 1);
                 byReceiveBuf[nBytes] = '\0';
                 if(nBytes == 0)  //没接收到数据
                 {
```

```
                        continue;        //继续循环
                    }
                    //开始对收到的字符串进行处理
                    CString str;
                    //读取接收文本框中原来的字符
((CEdit*)Pthis->GetDlgItem(IDC_EDIT_RECEIVE))->GetWindowTextW(str);
                    CString str_new;
                    //把新接收到的字符转换为CString类型
                    str_new = byReceiveBuf;
                    //连接原来的字符和新的字符
                    str += str_new;
                    //设置接收文本框的字符
((CEdit*)Pthis->GetDlgItem(IDC_EDIT_RECEIVE))->SetWindowTextW(str);
            }
    }
    return 0;
}
```

（4）编写写串行端口的写函数WritePort

在UartDlg.h文件中添加WritePort函数声明与串口控制驱动，如图6-45所示。

```
public:
    HANDLE m_hcom;                      //串口控制驱动
  BOOL WritePort(const char *str);      //WritePort函数声明
```

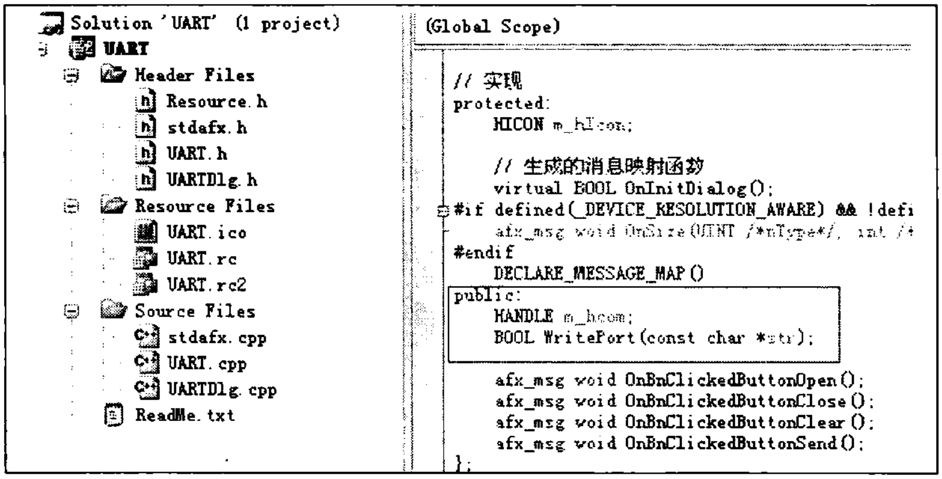



图6-45　添加WritePort函数声明与串口控制驱动

在该项目中添加名为WritePort的函数，实现部分的程序代码如下：

```
BOOL CUartDlg::WritePort(const char *str)
{
   DWORD dwNumBytesWritten;
   DWORD dwHaveNumWritten =0 ; //已经写入多少
   DWORD bufLen = strlen(str);
   //判断串行埠是否有效
   if(m_hcom == NULL || m_hcom == INVALID_HANDLE_VALUE)
   {
          return FALSE;
```

```
    }
    do
    {
            if(WriteFile(m_hcom,                          //串行埠句柄
                    str+dwHaveNumWritten,                 //被写数据缓冲区
                    bufLen - dwHaveNumWritten,            //被写数据缓冲区大小
                    &dwNumBytesWritten,                   /*函数执行成功后,返回实际向串行埠写的个数*/
                    NULL))                                //此处必须设置NULL
            {
                    dwHaveNumWritten = dwHaveNumWritten + dwNumBytesWritten;
                    //写入完成
                    if(dwHaveNumWritten == bufLen)
                    {
                            break;
                    }
                    Sleep(10);
            }
            else
            {
                    return FALSE;
            }
    }while(TRUE);
    return TRUE;
}
```

（5）给“Open”按钮添加程序代码

在“打开”按钮中添加打开串口、配置串口和启动接收线程的程序代码。这里打开串口使用CreateFile函数。以实验平台上的COM1口为例，在Resource View中依次展开“LED”→“Uart.rc”→“Dialog”，双击窗口中的“打开”按钮，添加如下代码：

```
//打开串行口
    m_hcom = CreateFile(_T("COM1:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,0,NULL);
    //检测是否打开串行口成功
    if(m_hcom != INVALID_HANDLE_VALUE && m_hcom != NULL)
    {
            DCB  dcb;
            dcb.DCBlength = sizeof(DCB);
            //配置串行端口参数
            GetCommState(m_hcom, &dcb);
            dcb.BaudRate = CBR_115200 ;                   //波特率
            dcb.ByteSize = 8;                             //数据位
            dcb.Parity = NOPARITY;                        //奇偶校验
            dcb.StopBits = ONESTOPBIT;                    //停止位
            if(!SetCommState(m_hcom, &dcb))
            {
                    //配置串行端口失败
                    MessageBox(_T("Set COM port fail!"));
                    return ;
            }
            //配置超时值
            COMMTIMEOUTS  cto;
            GetCommTimeouts(m_hcom,/&cto);
```

```
            cto.ReadIntervalTimeout = MAXDWORD;
            cto.ReadTotalTimeoutMultiplier = 0;
            cto.ReadTotalTimeoutConstant = 0;
            cto.WriteTotalTimeoutMultiplier = 10;
            cto.WriteTotalTimeoutConstant = 1000;
            if(!SetCommTimeouts(m_hcom, &cto))
            {
                    //不能设置超时参数
                    MessageBox(_T("Set COM timeout fail!"));
                    return ;
            }
    }
    else
    {
            MessageBox(_T("Open COM fail!"));
            return ;
    }
    //串行口打开有效,启动读串行口线程
    DWORD m_dwThreadID;
    HANDLE m_hThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)ReadThread,this,0, &m_dwThreadID);
    CloseHandle(m_hThread);
```

（6）给“Close”按钮添加关闭串行端口的程序代码

回到Dialog，在窗体中双击“关闭”按钮并添加如下代码：

```
void CUartDlg::OnBnClickedButtonClose()
{
    //TODO: Add your control notification handler code here
    if(m_hcom)
    {
            CloseHandle(m_hcom);
            m_hcom = NULL;
    }
}
```

（7）给“Clear”按钮添加清空接收文本框的程序代码

```
void CUartDlg::OnBnClickedButtonClear()
{
    //TODO: Add your control notification handler code here
    ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetWindowTextW(_T(""));
}
```

（8）给“Send”按钮添加发送信息的程序代码

由于Windows CE平台上使用的都是Unicode编码的字符，所以需要转换成char型。

```
void CUartDlg::OnBnClickedButtonSend()
{
    //TODO: Add your control notification handler code here
CString str_send;
    //获取文本框中的字符
    ((CEdit*)GetDlgItem(IDC_EDIT_SEND))->GetWindowTextW(str_send);
    //判断字符串是否为空
    if(str_send.IsEmpty())
    {
```

```
        return;         //字符串为空,返回
    }
    char send[1024];
    //把宽字符转换为字符型
    wcstombs(send,str_send,1024);
    //发送信息
    if(!WritePort(send))
    {
        MessageBox(_T("Write Port fail"));
    }
}
```

（9）编译程序

在Visual Studio 2005上方工具栏中间的编译方式中选择“Release”，在SDK下拉菜单中选择“DMA6410（ARMV4I）”。

配置好后，依次选择“Build”→“Rebuild Uart”，开始编译程序。编译成功后即生成了可执行文件Uart.exe。该文件在文件夹“我的文档\Visual Studio 2005\Projects\Uart\Uart\DMA6410（ARMV4I）\Release”中。为了验证串口通信程序是否正确，需要将Uart.exe复制到开发平台DMA-6410XP中。

（10）验证程序

打开平台开关，启动Windows Embedded CE 6.0系统。启动后，PC上的ActiveSync软件显示连接成功，双击PC桌面右下角任务栏中的图标，打开ActiveSync软件。在“Microsoft ActiveSync”窗口中单击“浏览”，打开开发平台文件夹，在PC的路径为“我的文档\Visual Studio 2005\Projects\Uart\Uart\DMA6410（ARMV4I）\Release”目录中找到编译好的Uart.exe文件，将该文件复制到开发平台的NANDFLASH文件夹中，然后操作开发平台，在NANDFLASH文件夹中打开Uart.exe，双击运行程序。然后，用串口线将PC与实验平台的UART1串口连接起来，接着在PC桌面上打开DNW串口通信工具，如图6-46所示。

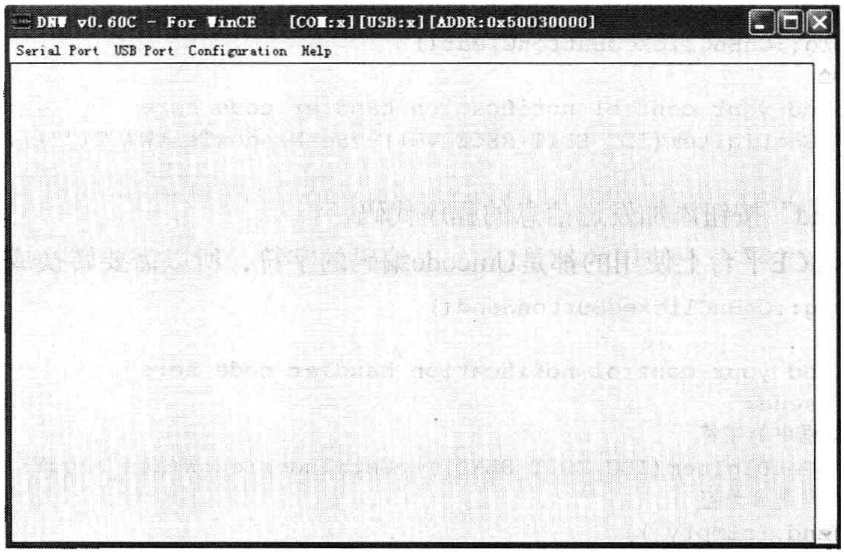



图6-46 DNW串口通信工具

依次单击“Serial Port”→“Connect”连接串口，然后执行已经复制到开发平台中的Uart.exe程序。接着单击“打开”按钮，启动串口通信。在PC的DNW中敲击键盘，即可在开发平台的Uart程序上面的文本框中接收到PC键盘发送的字符，即验证了计算机成功向开发平台发送数据。接下来在WINDOWS CE端下面的文本框中输入任意字符后单击“发送”按钮，在DNW中将显示输入框中的字符。

## 6.6　GPRS接口及驱动程序

在DMA-6410XP实验平台上插入GPRS模块。采用Visual Studio 2005编写GPRS驱动程序，在Windows Embedded CE 6.0下实现GPRS/GSM模块和DMA-6410XP实验平台拨打电话、发送短信功能，并将结果显示在LCD屏幕上。

### 6.6.1　GPRS/GSM工作原理

GPRS（General Packet Radio Service，整合封包无线电服务）是一种基于GSM系统的无线电封包交换技术，提供点对点的、广域的无线IP连接。相对原来GSM的拨号方式的电路交换数据传送方式，GPRS是封包交换技术。简单来讲，GPRS是一项高速数据传输的技术，方法是以“封包”的形式传送数据到用户手上。虽然GPRS是作为现有GSM网络向第三代移动通信的过渡技术，但是它在许多方面都具有显著优势。

GPRS有Um接口和GSM的移动接口。Um接口上的通信协议有5层，自下而上依次为物理层、MAC（Media Access Control）层、LLC（Logical Link Control）层、SNDC（SubNetwork Dependant Convergence）层和网络层。

GPRS网络是基于现有的GSM网络来实现的。在现有的GSM网络中需要增加一些节点，如GGSN（Gateway GPRS Supporting Node，GPRS网关支持节点）和SGSN（Serving GSN，GPRS服务支持节点）。GSN是GPRS网络中最重要的网络节点。GSN具有移动路由管理功能，它可连接各种类型的数据网络，并可连接到GPRS寄存器。GSN可完成移动终端和各种数据网络之间的数据传送和格式转换。GSN有两种类型：SGSN（Serving GSN，服务GSN）和GGSN（Gateway GSN，网关GSN）。SGSN的主要作用是记录移动终端的当前位置信息，并且在移动终端和GGSN之间完成移动分组数据的发送和接收；GGSN主要起网关作用，它可以与多种不同的数据网络连接，如ISDN、PSPDN和LAN等。

GPRS工作时是通过路由管理来进行寻址和建立数据连接的，而GPRS的路由管理表现在以下3个方面：

1）移动终端发送数据的路由建立。

2）移动终端接收数据的路由建立。

3）移动终端处于漫游时数据路由的建立。

对于第一种情况，当移动终端产生一个PDU（分组数据单元）时，这个PDU经过SNDC层处理，称为SNDC数据单元，然后经过LLC层处理为LLC帧，通过空中接口送到GSM网络中移

动终端所处的SGSN。SGSN把数据送到GGSN，GGSN把收到的消息进行解包处理，转换为可在公用数据网中传送的格式（如PSPDN的PDU），最终送给公用数据网的用户。对于第二种情况，当一个公用数据用户传送数据到移动终端时，通过数据网的标准协议建立数据网GGSN之间的路由。数据网用户发出的数据单元（如PSPDN中的PDU），通过建立好的路由把数据单元PDU送给GGSN，而GGSN在把PDU送给移动终端所在的SGSN时，GSN把PDU封装成SNDC数据单元，再经过LLC层处理LLC帧单元，最终通过空中接口送给移动终端。第三种情况是指一个数据网用户传送数据给一个正在漫游的移动用户，在这种情况下，数据传送必须首先经过归属地的GGSN，然后送到移动用户。

## 6.6.2 AT命令

下面介绍几个与GPRS模块有关的AT命令。

1）AT：测试命令。如果模块已正常工作，则返回OK。

2）ATE0：关闭“回声”。如果不使用该命令，当向模块发送“AT”时，则收到“AT”和“OK”两条信息；当使用该命令后，将只收到“OK”信息，使用该命令后将不返回发送的是什么命令。

3）ATD+电话号码+；：电话呼出命令。例如，ATD10086；。注意，最后的“；”是必不可少的。该命令正常执行时返回OK，否则返回ERROR加上错误码。

4）ATH：切断当前的所有通信，可用于挂断当前的通话或拒绝接听来电。

5）ATA：接听来电。当有电话呼入时，使用该命令可以接听该来电。

6）AT+CLIP=1：设置显示来电信息。当有电话打来时，将会收到类似：+CLIP：“045186229710”，129，“”，“”，0这样的信息，这里的“045186229710”则是来电号码。

7）AT+CSCA？：获取短信中心的号码。在短信息收发的时候使用。

8）AT+CNMI=1，1，0，0，1：设置新短信息来临的存储方式。

9）AT+CMGS=信息长度：告诉模块要发送的信息的长度。在发送短信息的时候使用。

10）AT+CMGR=index：读取指定（index）的短信内容。用于读取短信息。

## 6.6.3 电话程序设计

GPRS分为两个应用程序：电话软件和短信软件。

先来看看电话软件，该软件主要实现拨出电话和显示呼入电话号码，以及接听来电功能。

（1）软件界面制作

首先制作如图6-47所示的电话软件界面。

这里使用了一个称作“Group Box”的控件，如图6-47中的“Call”和“Answer”。如果读者曾经使用过该控件，不知道是否遇到过这样的情况，原想实现如图6-47所示的界面，但当程序执行时，却发现文本框和按钮都不见了，或是只有某些控件可见。

解决这个问题的办法是设置这些控件的Table顺序。在添加控件的界面上，选中窗体或任意一个控件，按<Ctrl+D>组合键，这时将出现如图6-48所示的界面。

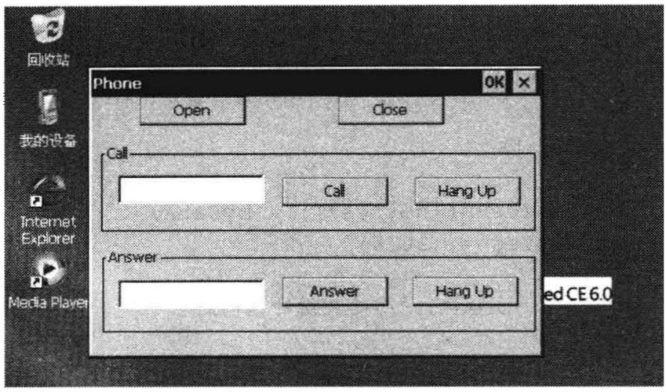



图6-47 电话软件界面

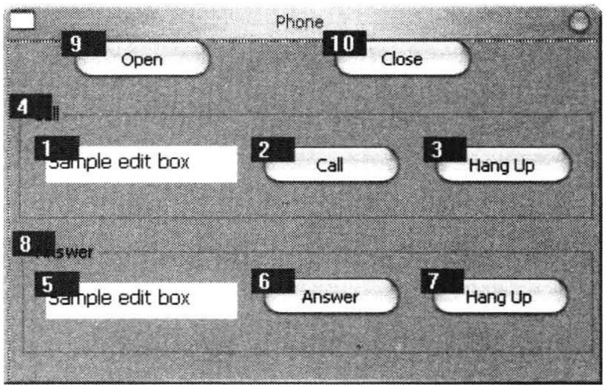



图6-48 设置控件Table顺序

图6-48中的数字表示这些控件的Table顺序，通过选择这些数字，可以更改它们的值，只要设置“Group Box”的Table顺序值大于在它“上面”显示的控件的值即可。可以尝试更改它们的Table顺序值，观察一下会有什么样的现象发生。

（2）实现读串行端口线程函数ReadThread

这个应用看起来是在操控电话模块，但实际上还是在对串行端口进行读/写，通过向串行端口发送不同的AT命令来使模块完成不同的功能。因此，这个程序和上一个UART程序有很多相似之处，如对串行端口的读/写操作。下面先来实现读串行端口的线程函数。

```
//接收串口数据线程
UINT ReadThread(LPVOID pParam)
{
    CPhoneDlg *Pthis = (CPhoneDlg*)pParam ;      /*转换参数为CUartDlg类型,以便以后可以
使用它的成员变量和函数等*/
    if(Pthis == NULL)      //检查参数是否有效
            return 0;
    BYTE   byData;
    DWORD   dwBytes;
    BYTE    byReceiveBuf[1024];  //缓冲区的大小
    int     nBytes= 0;
    DWORD evtMask;

    //设置字符监听信号
    SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
    //只要m_hcom有效,就一直循环
    while(Pthis->m_hcom != NULL && Pthis->m_hcom != INVALID_HANDLE_VALUE)
    {
            //等待收到串口信号
            if(WaitCommEvent(Pthis->m_hcom, &evtMask,0))
            {
                    //设置字符监听信号
                    SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
                    //清空接收数组
                    memset(byReceiveBuf,0,1024);
                    nBytes = 0;
                    do
```

```
                {
                        //读字符
                        if(ReadFile(Pthis->m_hcom, &byData,1, &dwBytes,0))
                        {
                                if(dwBytes == 1)
                                {
                                        byReceiveBuf[nBytes++] = byData;
                                        if(nBytes == 1024)
                                        {
                                                break;  //接收缓冲区已满
                                        }
                                }
                        }
                } while(dwBytes == 1);
                byReceiveBuf[nBytes] = '\0';
                if(nBytes == 0)  //没接收到数据
                {
                        continue;       //继续循环
                }
                TCHAR   szRecv[1024];
                mbstowcs(&szRecv[0], (char*)byReceiveBuf,1024);
                RETAILMSG(1, (TEXT("<%s"),szRecv));
                //开始对收到的字符串进行处理
                Pthis->ParseData((char*)byReceiveBuf);
        }
    }
    return 0;
}
```

（3）添加写串行端口函数WritePort

写串行端口函数Write Port代码如下。

```
BOOL CPhoneDlg::WritePort(const char *str)
{
    DWORD dwNumBytesWritten;
    DWORD dwHaveNumWritten =0 ; //已经写入多少
    DWORD bufLen = strlen(str);
    //判断串口是否有效
    if(m_hcom == NULL || m_hcom == INVALID_HANDLE_VALUE)
    {
            return FALSE;
    }
    do
    {
            if(WriteFile(m_hcom,                    //串口句柄
                    str+dwHaveNumWritten,           //被写数据缓冲区
                    bufLen - dwHaveNumWritten,      //被写数据缓冲区大小
                    &dwNumBytesWritten,             //函数执行成功后,返回实际向串口写的个数
                    NULL))                          //此处必须设置NULL
            {
                    dwHaveNumWritten = dwHaveNumWritten + dwNumBytesWritten;
                    //写入完成
                    if(dwHaveNumWritten == bufLen)
```

```
                {
                        break;
                }
                Sleep(10);
        }
        else
        {
                return FALSE;
        }
    }while(TRUE);
    return TRUE;
}
```

（4）给“Open”按钮添加程序代码

这个“Open”按钮实现的功能和UART的“Open”按钮类似，都是打开串行端口，并设置一些参数，然后启动读数据线程，只是这里多了一些功能，就是向串行端口发送两个命令：ATE0和AT+CLIP=1，程序代码如下所示：

```
//发送ATE0,关闭“回声”
    //注意,所有的命令使用\r\n结尾,表示一条命令结束
    WritePort("ATE0\r\n");
    /*延时1秒,这个很重要,不要连续的发送多条命令给模块,由于模块处理前一条命令需要时间,所以可能导致
不能处理紧接着的命令*/
    //延时是为了让模块有时间处理上一条命令,再发送下一条命令
    Sleep(1000);    //该函数的参数单位是毫秒,1000毫秒= 1秒
    //发送AT+CLIP=1,显示来电信息
    //只有设置了这个命令,才可以在串行端口收到的消息中解析出来电号码.
    WritePort("AT+CLIP=1\r\n");
```

需要注意的是，每条命令都以“\r\n”结尾，同时，两条命令之间的延时也很重要。

（5）给“Call”按钮添加程序代码

“Call”按钮是用来拨打电话的，其流程是先从文本框中取得号码，然后把获取到的号码组合成“ATD+号码+；”命令，最后把该命令发送出去。其程序代码如下：

```
CString str;
    //获取文本框中输入的电话号码
    ((CEdit*)GetDlgItem(IDC_EDIT_CALL))->GetWindowTextW(str);
    //把号码转换成命令格式“ATD+号码+; ”,注意这里的“; ”
    str = _T("ATD") + str + _T(";\r\n");
    char s[64];
    //转换字符
    wcstombs(s,str,64);
    //发送命令
    WritePort(s);
```

（6）给“Hang Up”按钮添加程序代码

“Hang Up”按钮的作用是关闭当前通话，只需要发送ATH命令即可。本例涉及的两个“Hang Up”按钮的程序代码完全一样。

（7）测试电话拨打功能

编译执行软件后，在文本框中输入想调用的电话号码，然后选择“Call”按钮。

（8）给ParseData函数添加程序代码

ParseData函数用来处理收到的串行端口消息。串行端口会收到哪些消息呢？每当给模块发送一条AT命令时，都会收到一条该命令执行后的返回信息，最常见的是OK、ERROR等。还有一些信息是由模块主动发送的，比如来电信息。当然，这个信息只有要求先给模块发送“AT+CLIP=1”这样的命令后才会出现，而在Open函数中已经发送了这个命令。

应该处理哪些消息呢？这些收到的信息都可以处理，但只处理来电信息。当有电话打入时，会收到类似：+CLIP：“075586229710”，129，“”，“”，0这样的信息，在这个函数中应该检测是否收到这样的信息，当收到这样的信息时，还要把信息中的来电号码解析出来，然后显示在文本框中。其程序代码如下：

```
void CPhoneDlg::ParseData(const char *data)
{
    //如果没有收到完成的信息,则不处理,而是等待收到后再处理
    if(!CheckData(data))
            return;
    //判断是否有来电号码的信息
    char *pfind = strstr(m_data, "+CLIP:");
    if(pfind != NULL)
    {
            pfind = strchr(pfind, '"');
            int i = 0;
            char number[32];
            pfind++;
            //把号码复制到number数组中
            while(i<31 && *pfind != '"')
            {
                    number[i] = *pfind;
                    i++;
                    pfind++;
            }
            number[i] = '\0';
            //将号码显示在文本框中
            CString str;
            str = number;
            ((CEdit*)GetDlgItem(IDC_EDIT_ANSWER))->SetWindowTextW(str);
            memset(m_data,0,sizeof(m_data));
    }
}
```

（9）测试显示来电号码

在给ParseData函数添加程序代码后，测试是否有实现预期的功能。当第一次调用该号码时，在文本框中显示号码，但当第二次拨打时，程序则异常退出。经多次测试后，发现有时正常，有时程序会退出，有时会显示来电号码，但没有完全显示，只显示了前几位。

根据上面描述的现象，发现应该是在ParseData函数中处理来电信息号码的程序代码有问题。通过断点除错的方法，发现当出现问题时，收到的“data”数据信息不是一条完整的信息，于是可在接收信息的线程函数中添加打印当前收到的信息的程序代码。

```
TCHAR szRecv[1024];
    mbstowcs(&szRecv[0], (char*)byReceiveBuf,1024);
```

```
RETAILMSG(1, (TEXT("<%s"),szRecv));
```

通过上面几行程序代码，可以在打印信息中清楚地看到一条信息分几次收到的现象。

由于当初的ParseData函数中的程序代码是针对一条完整的信息来编写的，所以，当现在收到一条不完整的信息时，软件就会出现不同的异常情况。为了解决这个问题，在解析收到的信息前，决定先检查收到的是否为一条完整的信息。如果是一条完整的信息，则可以解析，否则就整合这些不全的信息，等到收到完整的信息后再来解析。

如何整合这些不全的信息呢？要想解决这个问题，应该先知道如何判断每条信息在正常情况下是如何结尾的。每条信息都是以“\r\n”字符结束的。

这里使用CheckData函数来处理它们，把整合好的数据放到m_data数组中，最后去解析m_data数组中的数据。该函数的程序代码如下：

```
//判断该数据的最后两位是否是\r\n
    if(data[strlen(data)-2] == '\r' && data[strlen(data)-1] == '\n')
    {
            if(m_data[0] == 0)//这条信息是第一次收到,且全部收到
            {
                    strcpy(m_data,data);
                    return TRUE;
            }
            else    //这条信息不是第一次收到,但是全部接收完
            {
                    strcat(m_data,data);
                    return TRUE;
            }
    }
    else    //没有收到一条信息的结尾
    {
            if(m_data[0] == 0)//这条信息是第一次收到,且没有全部收到
            {
                    strcpy(m_data,data);
                    return FALSE;
            }
            else    //这条信息不是第一次收到,并且没有全部接收完
            {
                    strcat(m_data,data);
                    return FALSE;
            }
    }
```

在添加该函数后，修改ParseData函数中的前几行程序代码，如下：

```
//如果没有收到完成的信息,不处理,等到收到后再处理
    if(!CheckData(data))
            return;
    //判断是否有来电号码的信息
    char *pfind = strstr(m_data, "+CLIP:");
```

这里使用的是m_data数组，并且在每次解析完后要把m_data中的资料全部清空，因为在CheckData函数中有“if ( m_data[0] == 0 )”比较语句。

再来编译执行该程序，会发现来电号码显示在文本框中。

（10）给“Answer”按钮添加接听电话的程序代码

有了来电显示后，可以接听该电话，在“Answer”按钮中添加下面这行程序代码即可：

```
WritePort("ATA\r\n");
```

至此，电话程序软件设计已基本完成，最后可综合测试一下。

## 6.6.4 短信程序设计

短信的收发软件和电话软件十分类似，只是要发送和处理的接收信息不同，下面进行具体介绍。

（1）实现软件界面

首先，实现类似如图6-49所示的软件界面。

图6-49中的“Number”和“Text”使用的是Static Text控件，它们所对应的文本框分别用于输入电话号码和短信内容。

（2）向项目中添加文件

在该软件中将使用一些具有字符转换功能的函数，这些函数分别存储在Sms.h和Sms.cpp两个文件中，下面先把这两个文件加入到项目中。

首先，把这两个文件复制到当前项目，即与MessageDlg.h和MessageDlg.cpp相同的文件夹。

然后，找到Solution Explorer视图窗口，如图6-50所示。

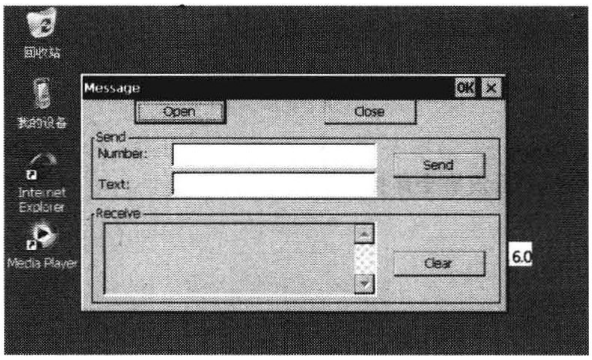



图6-49 短信软件界面

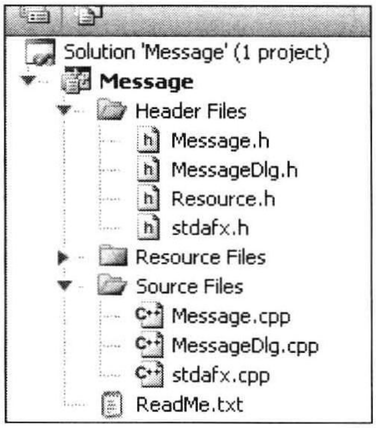



图6-50 Solution Explorer视图窗口

接着，在“Header Files”文件夹上单击鼠标右键，在弹出的快捷菜单中选择“Add”→“Existing Item…”，这时将出现一个打开文件的对话框，选择刚才复制的Sms.h文件，这样就可以添加该文件到项目中。

以类似的方法给“Source Files”文件夹添加Sms.cpp文件。最后，还需要在MessageDlg.h文件中加入以下引用程序代码：

```
#include "Sms.h"
```

（3）实现读串行端口线程函数和写串行端口函数

ReadThread函数和WritePort函数的实现与电话软件类似，代码如下：

```
//接收串口数据线程
UINT ReadThread(LPVOID pParam)
{
   CMessageDlg *Pthis = (CMessageDlg*)pParam ;  /*转换参数为CUartDlg类型,以便以后可以
使用它的成员变量和函数等*/
   if(Pthis == NULL)      //检查参数是否有效
          return 0;
   BYTE   byData;
   DWORD   dwBytes;
   BYTE    byReceiveBuf[1024];  //缓冲区的大小
   int     nBytes= 0;
   DWORD   evtMask;

   //设置字符监听信号
   SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
   //只要m_hcom有效,就一直循环
   while(Pthis->m_hcom != NULL && Pthis->m_hcom != INVALID_HANDLE_VALUE)
   {
          //等待收到串口信号
          if(WaitCommEvent(Pthis->m_hcom, &evtMask,0))
          {
                 //设置字符监听信号
                 SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
                 //清空接收数组
                 memset(byReceiveBuf,0,1024);
                 nBytes = 0;
                 do
                 {
                        //读字符
                        if(ReadFile(Pthis->m_hcom, &byData,1, &dwBytes,0))
                        {
                               if(dwBytes == 1)
                               {
                                      byReceiveBuf[nBytes++] = byData;
                                      if(nBytes == 1024)
                                      {
                                             break;  //接收缓冲区已满
                                      }
                               }
                        }
                 } while(dwBytes == 1);
                 byReceiveBuf[nBytes] = '\0';
                 if(nBytes == 0)  //没接收到数据
                 {
                        continue;      //继续循环
                 }
                 TCHAR   szRecv[1024];
                 mbstowcs(&szRecv[0], (char*)byReceiveBuf,1024);
```

```
                RETAILMSG(1, (TEXT("<%s"),szRecv));
                //开始对收到的字符串进行处理
                Pthis->ParseData((char*)byReceiveBuf);
            }
    }
    return 0;
}
BOOL CMessageDlg::WritePort(const char *str)
{
    DWORD dwNumBytesWritten;
    DWORD dwHaveNumWritten =0 ; //已经写入多少
    DWORD bufLen = strlen(str);
    //判断串口是否有效
    if(m_hcom == NULL || m_hcom == INVALID_HANDLE_VALUE)
    {
            return FALSE;
    }
    do
    {
            if(WriteFile(m_hcom,                    //串口句柄
                str+dwHaveNumWritten,               //被写数据缓冲区
                bufLen - dwHaveNumWritten,          //被写数据缓冲区大小
                &dwNumBytesWritten,                 //函数执行成功后,返回实际向串口写的个数
                NULL))                              //此处必须设置NULL
            {
                dwHaveNumWritten = dwHaveNumWritten + dwNumBytesWritten;
                //写入完成
                if(dwHaveNumWritten == bufLen)
                {
                        break;
                }
                Sleep(10);
            }
            else
            {
                return FALSE;
            }
    }while(TRUE);
    return TRUE;
}
```

（4）编写“Open”按钮程序代码

“Open”按钮实现的功能是打开串行端口、配置串行端口和启动线程的程序代码。需要注意的是，最后初始化模块的命令不同，这里发送的是AT+CNMI= 1，1，0，0，1和AT+CSCA?命令，分别用于设置接收到的短信的存储方式和获取短信中心的号码，程序如下。

```
void CMessageDlg::OnBnClickedButtonOpen()
{
    //TODO: Add your control notification handler code here
```

```
//打开串口
m_hcom = CreateFile(_T("COM1:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,0,NULL);
//检测是否打开串口成功
if(m_hcom != INVALID_HANDLE_VALUE && m_hcom != NULL)
{
        DCB  dcb;
        dcb.DCBlength = sizeof(DCB);
        //配置串口参数
        GetCommState(m_hcom, &dcb);
        dcb.BaudRate = CBR_115200 ;     //波特率
        dcb.ByteSize = 8;               //数据位
        dcb.Parity = NOPARITY;          //奇偶校验
        dcb.StopBits = ONESTOPBIT;      //停止位

        if(!SetCommState(m_hcom, &dcb))
        {
                //配置串口失败
                MessageBox(_T("Set COM port fail!"));
                return ;
        }
        //配置超时值
        COMMTIMEOUTS  cto;
        GetCommTimeouts(m_hcom, &cto);
        cto.ReadIntervalTimeout = MAXDWORD;
        cto.ReadTotalTimeoutMultiplier = 0;
        cto.ReadTotalTimeoutConstant = 0;
        cto.WriteTotalTimeoutMultiplier = 10;
        cto.WriteTotalTimeoutConstant = 1000;
        if(!SetCommTimeouts(m_hcom, &cto))
        {
                //不能设置超时参数
                MessageBox(_T("Set COM timeout fail!"));
                return ;
        }
}
else
{
        MessageBox(_T("Open COM fail!"));
        return ;
}
//串口打开有效,启动读串口线程
DWORD m_dwThreadID;
HANDLE m_hThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)ReadThread,this,0, &m_dwThreadID);
CloseHandle(m_hThread);

//发送ATE0,关闭“回声”
//注意,所有的命令使用\r\n结尾,表示一条命令结束
WritePort("ATE0\r\n");
//延时1秒,这个很重要,不要连续地发送多条命令给模块,由于模块处理前一条命令需要时间,所以可能导致
//不能处理紧接着的命令
```

```
    //延时是为了让模块有时间处理了上一条命令,再发送下一条命令
    Sleep(1000);    //该函数的参数单位是毫秒
    //发送AT+CNMI=1,1,0,0,1,设置收到的短信息的存储方式
    WritePort("AT+CNMI=1,1,0,0,1\r\n");
    Sleep(1000);
    //发送AT+CSCA?,获取短信中心的号码
    WritePort("AT+CSCA?\r\n");
}
```

（5）在ParseData函数中编写处理AT+CACA？命令返回信息的程序代码

相关代码如下所示。

```
Void CMessageDlg::ParseData(const char *data)
{
    //如果没有收到完成的信息,不处理,等到收到后再处理
    if(!CheckData(data))
            return;
    //处理短信息中心号码
    if(ParseStr(m_data, "+CSCA:"))
    {
            int i = 0;
            //移动指针到号码开始位置
            char *pfind = strchr(m_data, '"');
            pfind += 2;
            //复制号码
            while(i<31 && *pfind != '"')
            {
                    m_MSMCenterAddress[i] = *pfind;
                    i++;
                    pfind++;
            }
            m_MSMCenterAddress[i] = '\0';
            memset(m_data,0,sizeof(m_data));
            return;
    }

    //发送短信息内容
    if(ParseStr(m_data, ">") && m_bsending)
    {
            WritePort(m_MSM_PDU);
            memset(m_data,0,sizeof(m_data));
            return;
    }

    //收到短信发送成功的返回信息
    if(ParseStr(m_data, "+CMGS:") && m_bsending)
    {
            m_bsending = false;
            CString str;
            //取得原来文本框中的字符
```

```
        ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->GetWindowTextW(str);
        //连接字符串
        str = str + _T("\r\n信息发送完毕");
        //显示字符串
        ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetWindowTextW(str);
        //把显示的焦点设置到文本最后
        ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetSel(str.GetLength(),str.GetAllocLength());
        memset(m_data,0,sizeof(m_data));
        return;
}

//在发送短信的时候收到ERROR返回码
if(ParseStr(m_data, "ERROR") && m_bsending)
{
        m_bsending = false;
        CString str;
        ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->GetWindowTextW(str);
        str = str + _T("\r\n信息发送失败!");
        ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetWindowTextW(str);
        ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetSel(str.GetLength(),str.GetAllocLength());
        memset(m_data,0,sizeof(m_data));
        return;
}

//收到新短信的提示信息
if(ParseStr(m_data, "+CMTI:"))
{
        ParseCMTI(m_data);
        memset(m_data,0,sizeof(m_data));
        return;
}

//读取短信(AT+CMGR=index)命令的返回信息,其中包括了短信的时间、号码和内容等
if(ParseStr(m_data, "+CMGR:"))
{
        char *pfind = strstr(m_data, "+CMGR:");
        pfind = strstr(pfind, "\r\n");
        pfind += 2;

        SM_PARAM readSmParam;
        memset(&readSmParam,0,sizeof(SM_PARAM));
        //解码短信,把解码后的信息放到readSmParam中
        gsmDecodePdu(pfind, &readSmParam);

        CString str_timer;
        CString str_number;
        CString str_text;
        //获取时间
        char timer[25];
        timer[0] = '2';
```

```
            timer[1] = '0';
            timer[2] = readSmParam.TP_SCTS[0];
            timer[3] = readSmParam.TP_SCTS[1];
            timer[4] = '.';
            timer[5] = readSmParam.TP_SCTS[2];
            timer[6] = readSmParam.TP_SCTS[3];
            timer[7] = '.';
            timer[8] = readSmParam.TP_SCTS[4];
            timer[9] = readSmParam.TP_SCTS[5];
            timer[10] = ' ';
            timer[11] = ' ';
            timer[12] = ' ';
            timer[13] = readSmParam.TP_SCTS[6];
            timer[14] = readSmParam.TP_SCTS[7];
            timer[15] = ':';
            timer[16] = readSmParam.TP_SCTS[8];
            timer[17] = readSmParam.TP_SCTS[9];
            timer[18] = ':';
            timer[19] = readSmParam.TP_SCTS[10];
            timer[20] = readSmParam.TP_SCTS[11];
            timer[21] = '\0';
            str_timer = timer;
            //获取号码
            str_number = readSmParam.TPA;

            //转换信息正文
            TCHAR text[128];
            memset(text,0,sizeof(text));
            mbstowcs(text,(char *)(readSmParam.TP_UD),strlen((char *)(readSmParam.TP_UD)));
            str_text = text;
            //把所有的信息内容显示在文本框中
            CString str;
            ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->GetWindowTextW(str);
            str = str + _T("\r\n收到新信息");
            str += _T("\r\n时间: ") + str_timer + _T("\r\n号码: ") + str_number + _T("\r\n") + str_text;
            ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetWindowTextW(str);
            ((CEdit*)GetDlgItem(IDC_EDIT_RECEIVE))->SetSel(str.GetLength(),str.GetAllocLength());
            memset(m_data,0,sizeof(m_data));
            return;
        }
        memset(m_data,0,sizeof(m_data));
    }
```

通过对串行端口打印信息的观察，发现这里的串行端口信息同样有一条返回信息分几次收到的情况发生，所以先实现了CheckData函数。

当对模块发送AT+CSCA?命令后，模块会返回当前的短信息中心的号码，其格式类似：+CSCA: “+86138007555XX”，145，这里的86138007555XX即是短信息中心号码，在发送短信时需要使，把它提取出来存储在m_MSMCenterAddress数组中，注意，这里的“+”不用提取。

（6）在“send”按钮中发送短信

发送短信使用的PDU模式的发送方式，发送短信操作可分为以下几个部分：

1）获取发送信息文本框和电话号码框的内容。

2）填充要发送的短信的信息结构体（SM_PARAM）。

3）对信息结构体进行编码转换，并在信息末尾加上Ctrl-Z结束符。

4）发送命令“AT+CMGS=信息长度”，告诉模块要发送的信息的长度。

5）当模块收到“AT+CMGS=信息长度”命令后，准备好发送信息后会返回“>”信息。

6）收到“>”符号后，发送编码转换后的结构体信息（向串行端口写入m_MSM_PDU中的数据）。

7）在模块成功发送信息后，会返回类似“+CMGS：index”这样的信息。处理该信息，显示成功发送信息，完成信息的发送。

以上步骤1～步骤4在“Send”按钮中完成，然后到ParseData函数中去等待“>”返回信息。由于这里的“>”返回信息没有“\r\n”的结尾信息，因此，在CheckData中，如果还是使用原来的判断方式来处理的话，那么将一直等在那里，为此，可增加单独的处理程序代码：

```
//该命令不以\r\n结尾,单独处理
   if(ParseStr(data, ">"))
   {
          strcpy(m_data,data);
          return TRUE;
   }
```

在ParseData函数中，收到“>”返回信息后，把编码后的m_MSM_PDU数据写入串行端口。然后，等待接收到类似“+CMGS：index”这样的信息，接收到后表示发送成功。如果返回ERROR之类的信息，则表示失败。

（7）接收短信

想要把接收到的信息及时显示出来，其实现方法类似来电检测，可分为以下几个步骤。

1）对收到的每条串行端口信息进行检测，看是否接收到以“+CMTI：”开头的信息，类似：+CMTI: “SM”，23，其中“SM”表示短信的存储区，23表示序号，可以把这里的23从信息中提取出来。

2）把刚收到的短信内容读取出来，即使用命令AT+CMGR=index来获取短信内容，其中index是刚才从信息中提取出来的序号（23）。

3）接收到的短信内容，命令AT+CMGR=index的返回是以“+CMGR：”开头的信息。

4）把收到的短信内容进行处理和译码操作，并把译码后的短信内容显示到软件界面上。

此项功能可以参考ParseData函数中相关程序代码进行设计。

（8）CheckData中的一段程序代码的作用

在CheckData函数中还有如下一段程序代码：

```
/*判断是否是读短信的返回信息,因为读短信的返回信息中间有\r\n,在这里应该判断最后的OK  */
        if(ParseStr(m_data, "+CMGR"))
        {
                if(data[strlen(data)-4] == 'O' && data[strlen(data)-3] == 'K')
                {
                        return TRUE;
                }
                else
                {
                        return FALSE;
                }
        }
        else
        {
                return TRUE;
        }
```

上述这段程序代码是在读取短信内容时使用的，因为在读取短信的返回信息中间会有一个“\r\n”字符。如果还是以前的判断方式，当前信息刚好在这里被切断接收到，那么收到的信息将缺少短信的正文和日期时间等内容。因此，这里应该改为判断信息最后的“OK”结束字符。

（9）编译软件

最后，编译该短信软件。

## 6.7 直流电动机接口及驱动程序

在DMA-6410XP实验平台上，GPIO接口与直流电动机连接，将PWM信号转换为直流电平以控制直流电动机。采用Visual Studio 2005编写直流电动机驱动程序，在Windows Embedded CE 6.0下实现对直流电动机启动及停止的控制，并将结果显示在LCD屏幕上。

### 6.7.1 直流电动机驱动电路

电动机的速度与施加的电位成正比，输出转矩则与电流成正比。想要直流电动机高效执行，常见的方法是施加一个PWM（脉宽调制）方波，其通-断比率对应于所需速度。电动机起到一个低通滤波器作用，将PWM信号转换为有效直流电位。PWM驱动信号很常用，因为使用微处理器的控制器很容易产生PWM信号。虽然用精确的脉冲宽度可以调节电动机的速度，但在实际应用中，PWM频率却是可变的，设计时应对其进行优化，以防止电动机“颤抖”。如要使直流电动机反转，那么必须转换电动机中电流的方向，为此，大多数设计师采用排列成H形的4个开关装置。

由DMA-6410XP以及相关电路图可知，控制直流电动机的GPIO为GPF14（GPF14→nENCAM），涉及的电路图如图6-51所示。

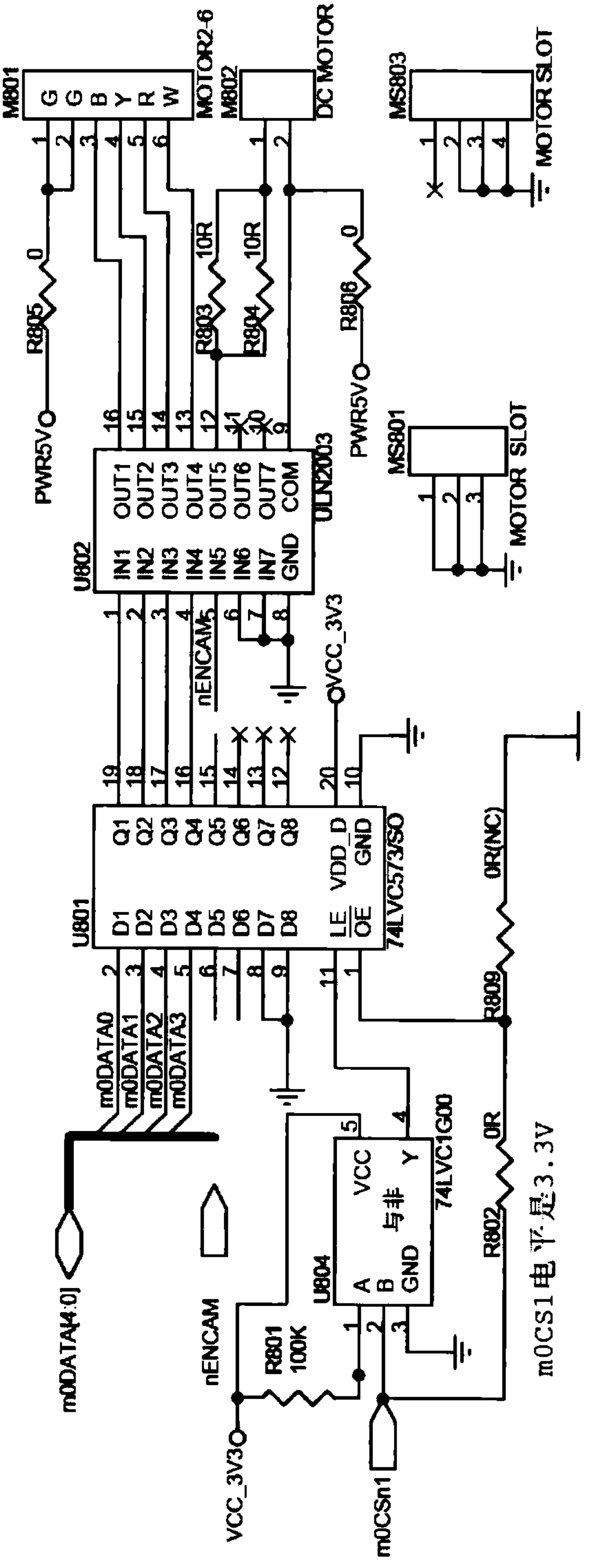



图6-51 直流电动机驱动电路

## 6.7.2 PWM定时器专用寄存器

下面介绍几种PWM定时器专用寄存器。

### 1. TCFG0

TCFG0（定时器配置寄存器0）及其位分配分别如表6-39和表6-40所示。

表6-39 TCFG0

| 寄存器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| TCFG0 | 0x7F006000 | 读/写 | 定时器配置寄存器0，可配置两个8位预定标器和“死区”长度 | 0x0000_0101 |

定时器输入时钟频率＝PCLK/{预定标器的值＋1}/{分频值}

{预定标器的值}＝0～255

{分频值}＝1，2，4，8，16，TCLK

表6-40 TCFG0的位分配

| TCFG0 | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Reserved | [31:24] | 读 | 保留 | 0x00 |
| Dead Zone Length | [23:16] | 读/写 | “死区”的长度 | 0x00 |
| Prescaler 1 | [15:8] | 读/写 | 预定标器1的值，用于定时器2、3和4 | 0x01 |
| Prescaler 0 | [7:0] | 读/写 | 预定标器0的值，用于定时器0和1 | 0x01 |

### 2. TCFG1

TCFG1（定时器配置寄存器1）及其位分配分别如表6-41和表6-42所示。

表6-41 TCFG1

| 寄存器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| TCFG1 | 0x7F006004 | 读/写 | 定时器配置寄存器1，可控制5MUX和DMA0 | x0000_0000 |

表6-42 TCFG1的位分配

| TCFG1 | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Reserved | [31:24] | 读 | 保留位 | 0x00 |
| DMA mode | [23:20] | 读/写 | 选择DMA请求通道选择位：<br>0000：无选择 0001：INT0<br>0010：INT1 0011：INT2<br>0100：INT 3 0101：INT4 | 0x00 |
| Divider MUX 4 | [19:16] | 读/写 | 选择定时器4的MUX输入：<br>0000：1/1 0001：1/2<br>0010：1/4 0011：1/8<br>0100：1/16 0101：外部TCLK1 | 0x00 |
| Divider MUX 3 | [15:12] | 读/写 | 选择定时器3的MUX输入：<br>0000：1/1 0001：1/2<br>0010：1/4 0011：1/8<br>0100：1/16 0101：外部TCLK1 | 0x00 |

（续）

| TCFG1 | 位 | 读/写 | 描 述 | 初始状态 |
|---|---|---|---|---|
| Divider MUX 2 | [11:8] | 读/写 | 选择定时器2的MUX输入：<br>0000：1/1 0001：1/2<br>0010：1/4 0011：1/8<br>0100：1/16 0101：外部TCLK1 | 0x00 |
| Divider MUX 1 | [7:4] | 读/写 | 选择定时器1的MUX输入：<br>0000：1/1 0001：1/2<br>0010：1/4 0011：1/8<br>0100：1/16 0101：外部TCLK1 | 0x00 |
| Divider MUX 0 | [3:0] | 读/写 | 选择定时器0的MUX输入：<br>0000：1/1 0001：1/2<br>0010：1/4 0011：1/8<br>0100：1/16 0101：外部TCLK1 | 0x00 |

### 3. TCON

TCON（定时控制寄存器）及其位分配分别如表6-43和表6-44所示。

表6-43 TCON

| 寄存器 | 地 址 | 读/写 | 描 述 | 复位值 |
|---|---|---|---|---|
| TCON | 0x7F006008 | 读/写 | 定时器控制寄存器 | 0x0000_0000 |

表6-44 TCON的位分配

| TCON | 位 | 读/写 | 描 述 | 初始值 |
|---|---|---|---|---|
| Reserved | [31:23] | 读 | 保留 | 0x000d |
| Timer 4 Manual Update（note） | [21] | 读/写 | 确定定时器4的手动更新<br>0 = 无操作 1 = 更新TCNTB4 | 0x0 |
| Timer 4 Start/Stop | [20] | 读/写 | 确定定时器4的启动/停止<br>0 = 停止 1 = 开始定时器4 | 0x0 |
| Timer 3 Auto Reload on/off | [19] | 读/写 | 确定定时器3的自动加载开/关<br>0 = One-shot 1 =间隔模式（自动重载） | 0x0 |
| Timer 3 Output Inverter on/off | [18] | 读/写 | 确定定时器3的输出反转器开/关<br>0 = 逆变器关闭 1 = 逆变器开，用于TOUT3 | 0x0 |
| Timer 3 Manual Update（note） | [17] | 读/写 | 确定定时器3的手动更新<br>0 = 无操作 1=更新TCNTB3或CMPB3 | 0x0 |
| Timer 3 Start/Stop | [16] | 读/写 | 确定定时器3的启动/停止<br>0 = 停止 1 = 开始定时器3 | 0x0 |
| Timer 2 Auto Reload on/off | [15] | 读/写 | 确定定时器2的自动加载开/关<br>0 = One-shot 1 =间隔模式（自动重载） | 0x0 |
| Timer 2 Output Inverter on/off | [14] | 读/写 | 确定定时器2的输出反转器开/关<br>0 =逆变器关闭 1 =逆变器开，用于TOUT2 | 0x0 |
| Timer 2 Manual Update（note） | [13] | 读/写 | 确定定时器2的手动更新<br>0 =无操作 1 = 更新TCNTB2或TCMPB2 | 0x0 |

（续）

| TCON | 位 | 读/写 | 描　述 | 初始值 |
|---|---|---|---|---|
| Timer 2 Start/Stop | [12] | 读/写 | 确定定时器2的启动/停止<br>0 = 停止 1 =开始定时器2 | 0x0 |
| Timer 1 Auto Reload on/off | [11] | 读/写 | 确定定时器1的自动加载开/关<br>0 = One-shot 1 =间隔模式（自动重载） | 0x0 |
| Timer 1 Output Inverter on/off | [10] | 读/写 | 确定定时器1的输出反转器开/关<br>0 =逆变器关闭1 = 逆变器开，用于TOUT1 | 0x0 |
| Timer 1 Manual Update（note） | [9] | 读/写 | 确定定时器1的手动更新<br>0 =无操作1 = 更新TCNTB1或TCMPB1 | 0x0 |
| Timer 1 Start/Stop | [8] | 读/写 | 确定定时器1的启动/停止<br>0=停止1=开始定时1 | 0x0 |
| Reserved | [7:5] | 读/写 | 保留 | 0x0 |
| Dead Zone Enable | [4] | 读/写 | 确定“死区”的操作<br>0=禁止 1=使能 | 0x0 |
| Timer 0 Auto Reload on/off | [3] | 读/写 | 确定定时器0的自动加载开/关<br>0=One-shot 1=间隔模式（自动重载） | 0x0 |
| Timer 0 Output Inverter on/off | [2] | 读/写 | 确定定时器0的输出反转器开/关<br>0=逆变器关闭 1=逆变器开，用于TOUT0 | 0x0 |
| Timer 0 Manual Update（note） | [1] | 读/写 | 确定定时器0的手动更新<br>0 =无操作1 = 更新TCNTB0或TCMPB0 | 0x0 |
| Timer 0 Start/Stop | [0] | 读/写 | 确定定时器0的启动/停止<br>0 = 停止 1 = 开始定时器0 | 0x0 |

### 6.7.3 直流电动机底层驱动程序设计

具体程序代码在BSP的SRC\Drivers\DC_Motor目录中，如图6-52所示。

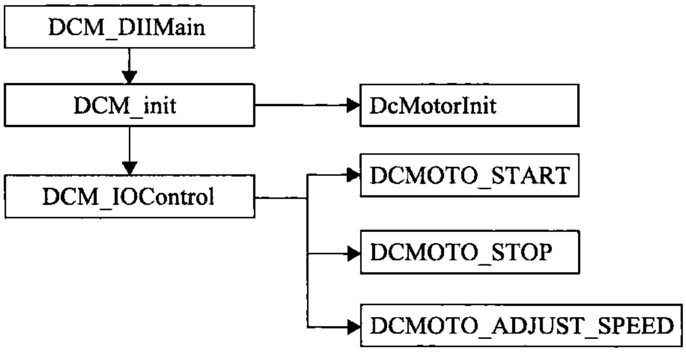



图6-52 流接口驱动

流驱动入口函数DCM_DllMain()主要接收进程和线程附加消息。DCM _Init()调用函数DcMotorInit()初始化直流电动机的状态。上层应用将直流电动机命令（开始、停止、调速）传给驱动的DCM_IOControl()函数，DCM_IOControl()则根据分支（开始、停止、调速）对直流电动机进行相应的操作。

DC_Motor有关程序代码在BSP中的SRC\Drivers\ DC_Motor目录下。

```
#include <windows.h>
#include <ceddk.h>
#include <S3c6410.h>

#include "s3c6410_pwm.h"
#include "s3c6410_gpio.h"


#define DCM_DBG_MSG  0
#define DCM_ERR_MSG  1

#define  MIN_SPEED        10
#define  MAX_SPEED        90


#define FREQ_PWM0         2000
变量
volatile S3C6410_GPIO_REG *v_pDcMotorGpioReg = NULL;
volatile S3C6410_PWM_REG *v_pDcMotorPwmReg = NULL;

UINT8 g_u8SpeedRatio;
UINT8 g_u8SpeedRatioSave;
BOOL RunFlg=FALSE;

typedef enum _DCMOTO_CTRL
{
   DCMOTO_ADJUST_SPEED=0x18,
   DCMOTO_STOP = 0x20,
   DCMOTO_START
}DCMOTO_CTRL;

void ConfigTout0();

void Set_Tout0_Pwm(UINT8 HiRatio)
{
   if(HiRatio > MAX_SPEED)
          HiRatio = MAX_SPEED;
   else if(HiRatio<MIN_SPEED)
          HiRatio = 0;

   RETAILMSG(DCM_DBG_MSG, (TEXT("++ Set_TOUT0_PWM ++\r\n")));
   g_u8SpeedRatioSave = HiRatio;

   ConfigTout0();

   v_pDcMotorPwmReg->TCON &= ~(0xf<<0); // 清除手动更新位,停止定时器0

   v_pDcMotorPwmReg->TCFG0 &= ~0xf;
   v_pDcMotorPwmReg->TCFG0 |= 0x3;       //设置定时器0/1预分频器的值为3
```

```
        v_pDcMotorPwmReg->TCFG1 &= ~0xf;
        v_pDcMotorPwmReg->TCFG1 |= 0x3;        //设置定时器0从UX输入为1/16

        v_pDcMotorPwmReg->TCNTB0 = (96000000)/(FREQ_PWM0*16);/*如果逆变器关闭,TCNT1<=TCMP1,
输出为高,TCNT1>TCMP1, 输出为低*/
        v_pDcMotorPwmReg->TCMPB0 = (v_pDcMotorPwmReg->TCNTB0*(MIN_SPEED +((MAX_SPEED - MIN_
SPEED)* HiRatio/100)))/100;

        v_pDcMotorPwmReg->TCON &= ~0xf;
        v_pDcMotorPwmReg->TCON |= 0x02;

        v_pDcMotorPwmReg->TCON &= ~0xf;        //启动定时器0
        v_pDcMotorPwmReg->TCON |= 0x09;        //逆变器关闭

        RETAILMSG(DCM_DBG_MSG, (TEXT("--Set_Tout0_Pwm --\r\n")));
    }


    void ConfigTout0()
    {
        RETAILMSG(DCM_DBG_MSG, (TEXT("++ConfigTout0\r\n")));

        v_pDcMotorGpioReg->GPFCON &= ~(0x3<<28);
        v_pDcMotorGpioReg->GPFCON |=  (0x2<<28);//设置GPF14为输出
        v_pDcMotorGpioReg->GPFPUD &= ~(0x3<<28);//禁止上拉

        RETAILMSG(DCM_DBG_MSG, (TEXT("--ConfigTout0\r\n")));
    }


    void CloseTout0()
    {
        RETAILMSG(DCM_DBG_MSG, (TEXT("++CloseTout0\r\n")));

        v_pDcMotorGpioReg->GPFCON &= ~(0x3<<28);
        v_pDcMotorGpioReg->GPFCON |= (0x1<<28);
        v_pDcMotorGpioReg->GPFDAT &= ~(0x1<<14);

        RETAILMSG(DCM_DBG_MSG, (TEXT("--CloseTout0\r\n")));
    }


    BOOL DcMotorInit(void)
    {
        DWORD dwIOSize;
        PHYSICAL_ADDRESS  IOPhyAdr= { S3C6410_BASE_REG_PA_GPIO,0 };
        PHYSICAL_ADDRESS  IOPhyPWM = { S3C6410_BASE_REG_PA_PWM,0};

        RETAILMSG(DCM_DBG_MSG, (TEXT("++DcMotorInit\r\n")));

        dwIOSize = sizeof(S3C6410_GPIO_REG);
```

```
        if(NULL == v_pDcMotorGpioReg)
                v_pDcMotorGpioReg = (volatile S3C6410_GPIO_REG *)MmMapIoSpace(IOPhyAdr,
(ULONG)dwIOSize,FALSE);

        dwIOSize = sizeof(S3C6410_PWM_REG);
        if(NULL == v_pDcMotorPwmReg)
                v_pDcMotorPwmReg = (volatile S3C6410_PWM_REG *)MmMapIoSpace(IOPhyPWM,
(ULONG)dwIOSize,FALSE);

        if((NULL==v_pDcMotorGpioReg) || (NULL==v_pDcMotorPwmReg))
        {
                RETAILMSG(DCM_ERR_MSG, (TEXT("InitMotorAddr Failer \r\n")));

                return FALSE;
        }
        else
                RETAILMSG(DCM_DBG_MSG, (TEXT("InitMotorAddr Success \r\n")));

        RETAILMSG(DCM_DBG_MSG, (TEXT("--DcMotorInit\r\n")));

        return TRUE;
    }


    DWORD DCM_Init(LPCTSTR pContext,LPCVOID lpvBusContext)
    {
        RETAILMSG(DCM_DBG_MSG, (TEXT("DCM_Init+++\r\n")));

        if(!DcMotorInit())
        {
                RETAILMSG(DCM_ERR_MSG, (TEXT("DcMotorInit Init failed !\r\n")));
        }

        g_u8SpeedRatio = 50;

        RETAILMSG(DCM_DBG_MSG, (TEXT("DCM_Init---\r\n")));

        return TRUE;
    }


    BOOL WINAPI DCM_DllMain(HANDLE hinstDll,ULONG Reason,LPVOID Reserved)
    {
        switch(Reason)
        {
                case DLL_PROCESS_ATTACH:
                        RETAILMSG(DCM_DBG_MSG,(TEXT("DC-MOTOR: DLL_PROCESS_ATTCH\r\n")));
                        break;

                case DLL_PROCESS_DETACH:
                        RETAILMSG(DCM_DBG_MSG,(TEXT("DC-MOTOR: DLL_PROCESS_DETACH\r\n")));
```

```
                    break;

            default:
                    break;
    }

    return TRUE;
}


BOOL DCM_IOControl(DWORD dwOpenContext,DWORD dwIoControlCode,LPBYTE lpInBuf,DWORD
                                        nInBufSize,LPBYTE lpOutBuf,DWORD nOutBuf-
                                        Size,LPDWORD lpBytesReturned
)
{
    RETAILMSG(DCM_DBG_MSG,(TEXT("DCM_IOControl+++\r\n")));

    switch(dwIoControlCode)
    {
            case DCMOTO_START:
                    RunFlg = TRUE;
                    Set_Tout0_Pwm(g_u8SpeedRatio);

                    RETAILMSG(DCM_DBG_MSG,(TEXT("DCMOTOR_IOControl: DCMOTO_START\r\n")));
                    break;
            case DCMOTO_ADJUST_SPEED:
                    if(NULL != lpInBuf)
                    {
                        g_u8SpeedRatio = (UINT8) (*lpInBuf);
                            Set_Tout0_Pwm(g_u8SpeedRatio);
                     }
                        else
                        {
                            RETAILMSG(DCM_ERR_MSG,(TEXT("DCMOTOR_IOControl:  Adjust
Speed Error\r\n")));
                            return FALSE;
                        }

                    break;
            case DCMOTO_STOP:
                  RETAILMSG(DCM_DBG_MSG,(TEXT("DCMOTOR_IOControl: DCMOTO_STOP\r\n")));
                  RunFlg = FALSE;
                                        CloseTout0();
                    break;
            default:
                  RunFlg = FALSE;
                  v_pDcMotorPwmReg->TCON &= ~(0xf<<0);

                    RETAILMSG(DCM_ERR_MSG,(TEXT("DCM_IOControl: default\r\n")));
                    break;
    }
```

```
   RETAILMSG(DCM_DBG_MSG,(TEXT("DCM_IOControl---\r\n")));

   return TRUE;
}
```

## 6.7.4 直流电动机驱动程序设计

下面介绍直流电动机驱动程序设计的步骤。

1）实现软件界面。

直流电动机软件界面如图6-53所示。

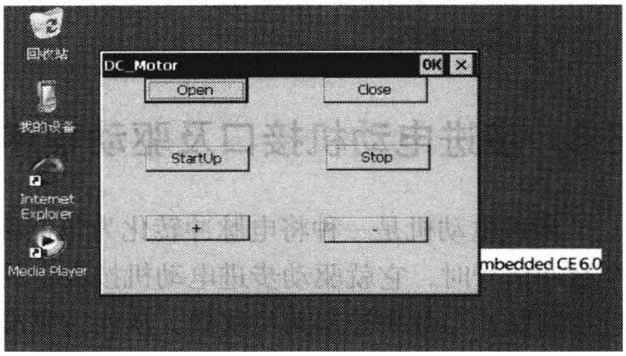



图6-53 直流电动机软件界面

"Open"按钮和"Close"按钮分别用于打开和关闭驱动。"StartUp"按钮和"Stop"按钮分别用于启动和停止电动机。"+"按钮和"–"按钮分别用于增加和减慢电动机的速度。

2）添加句柄的定义。

在DC_MotorDlg.cpp文件中做如下定义。

```
typedef enum _DCMOTO_CONTROL
{
   DCMOTO_ADJUST_SPEED=0x18,
   DCMOTO_STOP = 0x20,
   DCMOTO_START
}ENUM_DCMOTO_IOCONTROL;
```

3）给"Open"按钮添加打开驱动的程序代码。

```
m_hdcmotor = CreateFile(_T("DCM1:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,0,0);
   if(m_hdcmotor == INVALID_HANDLE_VALUE)
   {
           MessageBox(L"Open m_hdcmotor Driver error!",NULL,MB_OK);
   }
```

4）给"Close"按钮添加关闭驱动的程序代码。

```
if(m_hdcmotor != NULL)
   {
           CloseHandle(m_hdcmotor);
           m_hdcmotor = NULL;
   }
```

5）给"StartUp"按钮添加启动电动机的程序代码。

```
DeviceIoControl(m_hdcmotor,DCMOTO_START,NULL,NULL,NULL,NULL,NULL,NULL);
```

6）给"Stop"按钮添加停止电动机的程序代码。

```
DeviceIoControl(m_hdcmotor,DCMOTO_STOP,NULL,NULL,NULL,NULL,NULL,NULL);
```

7）给"+"按钮添加电动机加速的程序代码。

```
m_speed += 5;
   if(m_speed > 100)
   {
           m_speed = 100;
   }
   DeviceIoControl(m_hdcmotor,DCMOTO_ADJUST_SPEED, &m_speed,NULL,NULL,NULL,NULL,NULL);
```

8）给“-”按钮添加电动机减速的程序代码。

```
m_speed -= 5;
   if(m_speed < 5)
   {
          m_speed = 0;
   }
   DeviceIoControl(m_hdcmotor,DCMOTO_ADJUST_SPEED, &m_speed,NULL,NULL,NULL,NULL,NULL);
```

这样，就完成了该程序的编写，和前面的LED驱动程序类似，“StartUp”和“Stop”按钮的DeviceIoControl中不用使用第3个参数，而“+”按钮和“-”按钮需要使用第3个参数。

9）编译软件。

# 6.8 步进电动机接口及驱动程序

步进电动机是一种将电脉冲转化为角位移的执行机构。简单地讲，当步进驱动器接收到一个脉冲信号时，它就驱动步进电动机按设定的方向转动一个固定的角度（即步进角）。可以通过控制脉冲个数来控制角位移量，从而达到准确定位的目的，同时可以通过控制脉冲频率来控制电动机转动的速度和加速度，从而达到调速的目的。

在DMA-6410XP实验平台上配置有四相步进电动机，采用四路I/O进行脉冲分配，通过功率放驱动步进电动机。采用Visual Studio 2005编写步进电动机驱动程序，在Windows Embedded CE 6.0下实现对步进电动机顺时针旋转、逆时针旋转、加速、减速、启动及停止的控制，并将结果显示在LCD屏幕上。

## 6.8.1 步进电动机驱动电路

DMA-6410XP实验平台采用的是四相步进电动机，工作模式是四相四步，系统采用四路I/O进行脉冲分配，这四路I/O接口由CPU的Memory Port0扩充，通过功率放大后，进入步进电动机的各相绕组。

在四相四步的工作模式下，脉冲分配信号如表6-45所示，表中的A、B、C、D为步进电动机的四相。

表6-45 四相四步的脉冲分配信号

| 通电顺序 | 正转脉冲序列 | 反转脉冲序列 |
|---|---|---|
| 1 | 0011（AB） | 1001（DA） |
| 2 | 0110（BC） | 1100（CD） |
| 3 | 1100（CD） | 0110（BC） |
| 4 | 1001（DA） | 0011（AB） |

步进电动机驱动电路如图6-54所示。

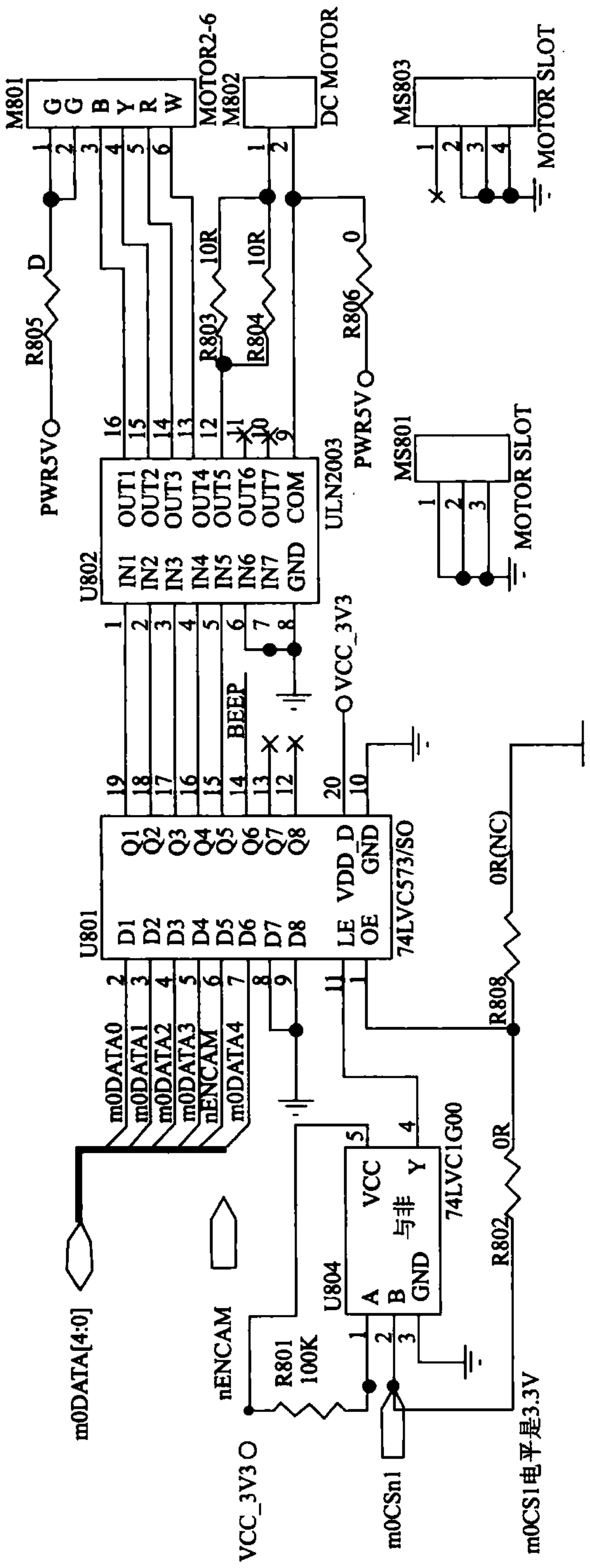



图6-54 步进电动机驱动电路

## 6.8.2 控制寄存器

SROM控制寄存器和SROM Bank1控制寄存器描述分别如表6-46～表6-49所示。

表6-46 SROM控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
| --- | --- | --- | --- | --- |
| SROM_BW | 0x70000000 | 读/写 | SROM总线宽度与等待控制 | 0x0000_000x |

表6-47 SROM控制寄存器的位分配

| SROM_BW | 位 | 描述 | 初始状态 |
| --- | --- | --- | --- |
| ByteEnable1 | [7] | 存储器Bank1的nWBE / nBE控制：<br>0=不使用UB/LB（XmWBE[1:0]是专用nWBE[1:0]）<br>1=使用UB/LB（XmWBE[1:0]是专用nBE[1:0]） | 0 |
| WaitEnable1 | [6] | 存储器Bank1等待允许控制：<br>0=WAIT不允许　　1=WAIT允许 | 0 |
| Reserved | [5] | 保留 | 0 |
| DataWidth1 | [4] | 存储器Bank1的数据线宽度控制：<br>0=8位　　1=16位 | 0 |

表6-48 SROM Bank1控制寄存器

| 寄存器 | 地址 | 读/写 | 描述 | 复位值 |
| --- | --- | --- | --- | --- |
| SROM_BC1 | 0x70000008 | 读/写 | SROM Bank1控制寄存器 | 0x000F_0000 |

表6-49 SROM Bank1控制寄存器的位分配

| SROM_BCn | 位 | 描述 | 初始状态 |
| --- | --- | --- | --- |
| Tacs | [31:28] | XmOCSn地址设置：<br>0000=0个时钟　0001=1个时钟<br>0010=2个时钟　0011=3个时钟<br>…<br>1100=12个时钟　1101=13个时钟<br>1110=14个时钟　1111=15个时钟 | 0000 |
| Tcos | [27:24] | nOE芯片选择设置：<br>0000=0个时钟　0001=1个时钟<br>0010=2个时钟　0011=3个时钟<br>…<br>1100=12个时钟　1101=13个时钟<br>1110=14个时钟　1111=15个时钟 | 0000 |
| Reserved | [23:21] | 保留 | 000 |
| Tacc | [20:16] | 访问周期：<br>00000=1个时钟　00001=2个时钟<br>00010=3个时钟　00011=4个时钟<br>…<br>11100=29个时钟　11101=30个时钟<br>11110=31个时钟　11111=32个时钟 | 01111 |

（续）

| SROM_BCn | 位 | 描　　述 | 初始状态 |
|---|---|---|---|
| Tcoh | [15:12] | 芯片选择维持nOE：<br>0000=0个时钟　0001=1个时钟<br>0010=2个时钟　0011=3个时钟<br>…<br>1100=12个时钟　1101=13个时钟<br>1110=14个时钟　1111=15个时钟 | 0000 |
| Tcah | [11:8] | Xm0CSn[x]地址维持时间：<br>0000=0个时钟　0001=1个时钟<br>0010=2个时钟　0011=3个时钟<br>…<br>1100=12个时钟　1101=13个时钟<br>1110=14个时钟　1111=15个时钟 | 0000 |
| Tacp | [7:4] | 页面方式访问周期：<br>0000=0个时钟　0001=1个时钟<br>0010=2个时钟　0011=3个时钟<br>…<br>1100=12个时钟　1101=13个时钟<br>1110=14个时钟　1111=15个时钟 | 0000 |
| Reserved | [3:2] | 保留 | |
| PMC | [1:0] | 页面方式配置：<br>00=正常（1个数据）01=4个数据<br>10=保留　　11=保留 | 00 |

## 6.8.3　步进电动机底层驱动程序设计

通过对有关电路图和寄存器的分析，当Xm0CSn1为低电位时（\WIN CE600\PLATFORM\DMA6410XP\SRC\OAL\OALLIB\init.c中InitializeBank()函数实现Xm0CSn1为低电位），m0DATA0[0:3]输出四相四步的脉冲分配信号来控制步进电动机，步进电动机的相序由ULN2003A芯片控制，如图6-55所示。

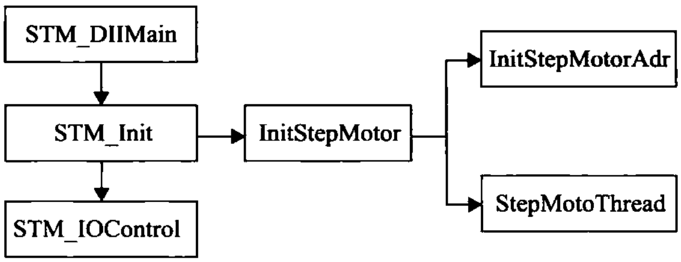



图6-55　步进电动机驱动

```
#include <windows.h>
#include <ceddk.h>
#include <S3c6410.h>
#include "s3c6410_gpio.h"


#define IO_CS1_PHY_ADR          0x18000000

#define STM_DBG_MSG   0
#define STM_ERR_MSG   1

// 控制参数
enum{
   SM_DIR_POSITIVE = 0x12,
   SM_DIR_NEGATIVE,
   SM_SPEED_INCREASE,
   SM_SPEED_DECREASE,
   SM_STOP,
   SM_START
};

#define  SM_MAX                    100
#define  SM_MIN                    10

// 启动步进电动机
#define NUM_WAVE_PIN       4
UINT8 StepWave_PDir[NUM_WAVE_PIN] = {0x01,0x04,0x02,0x08};
UINT8 StepWave_NDir[NUM_WAVE_PIN] = {0x08,0x02,0x04,0x01};

static UINT8 *v_pStepMotorData = NULL;


struct SM_CtrlParam
{
   HANDLE hSMStart;       //步进电动机启动事件句柄
   HANDLE hSMStop;        //步进电动机停止事件句柄
   BOOL bSMDir;           //步进电动机控制方向
   UINT uSMDelay;         //毫秒
   BOOL bSMRun;           //步进电动机运行状态
}g_StepMotor;


void StepMotorRun(UINT8 *pWave,DWORD dwSize,UINT uTime)
{
   DWORD dwCount;

   RETAILMSG(STM_DBG_MSG, (TEXT("++ StepMotorRun ++\r\n")));

   if(NULL != pWave)
```

```
        {
                for(dwCount = 0; dwCount < dwSize; dwCount++)
                {
                        if(uTime==SM_MAX)
                        {
                                *v_pStepMotorData &= ~(0x0f<<0);

                                return;
                        }
                        else
                        {
                                *v_pStepMotorData &= ~(0x0f<<0);
                                *v_pStepMotorData |= pWave[dwCount];

                                Sleep(uTime);
                        }
                }
        }
        else
        {
                RETAILMSG(STM_ERR_MSG, (TEXT("StepMotorRun Failed \r\n")));
                return;
        }
        RETAILMSG(STM_DBG_MSG, (TEXT("-- StepMotorRun --\r\n")));
    }


    BOOL InitStepMotorAdr()
    {
        PHYSICAL_ADDRESS IOPhySROMAdr = { IO_CS1_PHY_ADR,0 };
        PHYSICAL_ADDRESS IOPhySRomAddr = { S3C6410_BASE_REG_PA_SROMCON,0 };
        DWORD dwIOSize;
        static volatile S3C6410_SROMCON_REG *v_pSROMReg;

        dwIOSize = sizeof(UINT8 *);

        if(NULL == v_pStepMotorData)
        {
                v_pStepMotorData = (volatile UINT8 *)MmMapIoSpace(IOPhySROMAdr, (ULONG)
dwIOSize,FALSE);
                if(v_pStepMotorData)
                {
                        RETAILMSG(STM_DBG_MSG, (TEXT("InitStepMotorAdr : v_
pStepMotorData init successed\r\n")));
                }
                else
                {
                        RETAILMSG(STM_ERR_MSG, (TEXT("InitStepMotorAdr : v_
pStepMotorData init failed\r\n")));
```

```
                return FALSE;
            }
        }

    #if 0           //初始化CSN1的寄存器的功能: InitializeBank
        v_pSROMReg = (S3C6410_SROMCON_REG *)MmMapIoSpace(IOPhySRomAddr, (ULONG)dwIOSize,FALSE);

        v_pSROMReg->SROM_BW &= ~(0xF<<4);
        v_pSROMReg->SROM_BW |= ((0x1<<7)|  /* nWBE/nBE(for UB/LB)控制存储器的Bank1
(0=不使用UB/LB, 1=使用UB/LB)*/
                                (0x1<<6)| /*等待使能控制内存的Bank1(0=
                                           等待禁用,1=等待允许)*/
                                (0x1<<4));/*内存Bank1的数据总线宽度控
                                           制(0=8位,1=16位)*/

        v_pSROMReg->SROM_BC1 &= ~(0xFFFFFFFF<<0);
        v_pSROMReg->SROM_BC1 |=((0x2<<28)|        //Tacs, 在Xm0CSn[x]之前设置地址
                                (0x2<<24)|        //Tcos, 在nOE之前设置片选
                                (0x2<<16)|        //Tacc, 存储周期
                                (0x2<<12)|        //Tcoh, 对nOE保持片选
                                (0x2<<8) |        //Tcah,Xm0CSn[x]后地址保持时间
                                (0x0<<4) |        //Tacp, 页模式存取周期
                                (0x0<<0));        //PMC, 页面模式配置

        return TRUE;
    }


    DWORD StepMotorThread(void)
    {
        UINT8  *pWave = NULL;
        RETAILMSG(STM_DBG_MSG, (TEXT("beign Step-Motor thread\r\n")));

        while(TRUE)
        {
            if(WAIT_OBJECT_0 == WaitForSingleObject(g_StepMotor.hSMStart,INFINITE))
            {
                RETAILMSG(STM_DBG_MSG, (TEXT("Begin SM \r\n")));
                while(1)
                {
                    //调整方向
                    if(g_StepMotor.bSMDir)
                        pWave = StepWave_PDir;
                    else
                        pWave = StepWave_NDir;

                    //输出波形驱动步进电动机
                    StepMotorRun(pWave,NUM_WAVE_PIN,g_StepMotor.uSMDelay);
```

```
                        if(WAIT_OBJECT_0 == WaitForSingleObject(g_StepMotor.hSMStop,0) )
                        {
                                break;
                        }
                }
        }
        RETAILMSG(STM_DBG_MSG, (TEXT("Stop Step-Motor \r\n")));
    }

    return 0;
}


BOOL InitStepMotor()
{
    HANDLE  hSMDrv = NULL;

    if(!InitStepMotorAdr())
    {
        RETAILMSG(STM_ERR_MSG, (TEXT("Step-Motor InitStepMotorAdr Failed ! \r\n")));
        return FALSE;
    }
    else
    {
        RETAILMSG(STM_DBG_MSG, (TEXT("Step-Motor InitStepMotorAdr successed ! \r\n")));
    }

    hSMDrv  = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)StepMotorThread,NULL,0,NULL);

    //初始化g_StepMotor.
    g_StepMotor.hSMStart = CreateEvent(NULL,FALSE,FALSE,TEXT("Open Step-Motor"));
    g_StepMotor.hSMStop  = CreateEvent(NULL,FALSE,FALSE,TEXT("Stop Step-Motor"));
    g_StepMotor.bSMRun = FALSE;
    g_StepMotor.bSMDir = TRUE;
    g_StepMotor.uSMDelay = 50;

    *v_pStepMotorData &= ~(0x0f);

    if(hSMDrv)
    {
        CloseHandle(hSMDrv);
    }
    else
    {
        RETAILMSG(STM_ERR_MSG, (TEXT("Step-Motor CreateThread Failed ! \r\n")));
        return FALSE;
    }
```

```
    return TRUE;
}


BOOL STM_IOControl(DWORD dwOpenContext,DWORD dwIoControlCode,LPBYTE lpInBuf,
                   DWORD nInBufSize,LPBYTE lpOutBuf,DWORD nOutBufSize,
                   LPDWORD lpBytesReturned)
{
    RETAILMSG(STM_DBG_MSG,(TEXT("STM_IOControl+++\r\n")));

    switch(dwIoControlCode)
    {
        case SM_DIR_POSITIVE:
            g_StepMotor.bSMDir = TRUE;
            break;
        case SM_DIR_NEGATIVE:
            g_StepMotor.bSMDir = FALSE;
            break;
        case SM_SPEED_DECREASE:
            g_StepMotor.uSMDelay += 5;

            if(g_StepMotor.uSMDelay >SM_MAX)
            {
                g_StepMotor.uSMDelay = SM_MAX;
            }
            break;
        case SM_SPEED_INCREASE:
            g_StepMotor.uSMDelay -= 5;

            if(g_StepMotor.uSMDelay < SM_MIN)
            {
                g_StepMotor.uSMDelay = SM_MIN;
            }
            break;
        case SM_START:
            g_StepMotor.bSMRun = TRUE;
            SetEvent(g_StepMotor.hSMStart);
            break;
        case SM_STOP:
            if(g_StepMotor.bSMRun)
            {
                g_StepMotor.bSMRun = FALSE;
                SetEvent(g_StepMotor.hSMStop);
            }
            else
            {
                RETAILMSG(STM_DBG_MSG,(TEXT("The Step-Motor is not running ! \r\n")));
            }
            break;
```

```
        default:
                RETAILMSG(STM_ERR_MSG,(TEXT("STM_IOControl: default\r\n")));
                break;
    }
    RETAILMSG(STM_DBG_MSG,(TEXT("STM_IOControl---\r\n")));
    return TRUE;
}
```

## 6.8.4 步进电动机驱动程序设计

步进电动机驱动程序是根据流驱动接口实现的。流驱动入口函数DllMain()主要接收进程和线程附加消息。

STM _Init()函数完成物理地址到虚拟地址的转换，其中包括了创建步进电动机控制线程StepMotorThread()，以及InitStepMotorAdr()函数初始化。

InitStepMotorAdr()函数如下：

```
BOOL InitStepMotorAdr()
{
    PHYSICAL_ADDRESS IOPhySROMAdr = { IO_CS1_PHY_ADR,0 };
    PHYSICAL_ADDRESS IOPhySRomAddr = { S3C6410_BASE_REG_PA_SROMCON,0 };
    DWORD dwIOSize;
    static volatile S3C6410_SROMCON_REG *v_pSROMReg;
    dwIOSize = sizeof(UINT8 *);
    if(NULL == v_pStepMotor)
    {
            v_pStepMotor = (volatile UINT8 *)MmMapIoSpace(IOPhySROMAdr, (ULONG)dwIOSize,FALSE);
            if(v_pStepMotor)
            {
            RETAILMSG(STM_DBG_MSG, (TEXT("InitStepMotorAdr:v_pStepMotor init successed\r\n")));
            }
            else
            {
            RETAILMSG(STM_ERR_MSG, (TEXT("InitStepMotorAdr:v_pStepMotor init failed\r\n")));
                    return FALSE;
            }
    }
    return TRUE;
}
```

StepMotorThread()函数如下：

```
DWORD StepMotorThread(void)
{
    UINT8  *pWave = NULL;
    RETAILMSG(STM_DBG_MSG, (TEXT("beign Step-Motor thread\r\n")));
    while(TRUE)
    {
```

```
            if(WAIT_OBJECT_0 == WaitForSingleObject(g_StepMotor.hSMStart,INFINITE))
            {
                    RETAILMSG(STM_DBG_MSG, (TEXT("Begin SM \r\n")));
                    while(1)
                    {
                            //调整方向
                            if(g_StepMotor.bSMDir)
                                    pWave = StepWave_PDir;
                            else
                                    pWave = StepWave_NDir;
                            //输出波形驱动步进电动机
                            StepMotorRun(pWave,NUM_WAVE_PIN,g_StepMotor.uSMDelay);
    if(WAIT_OBJECT_0 ==WaitForSingleObject(g_StepMotor.hSMStop,0) )
                            {
                                    break;
                            }
                    }
            }
            RETAILMSG(STM_DBG_MSG, (TEXT("Stop Step-Motor \r\n")));
    }
    return 0;
}
```

STM_IOControl()函数接收应用程序传递来的参数，并根据参数来选择相应的条件分支，从而控制步进电动机的启动、停止、顺时针转、逆时针转、加速和减速等。具体程序代码在BSP的SRC\Drivers\Step_Motor目录下。

编译软件，步进电动机和直流电动机驱动程序相同，只是多了两个旋转方向的控制。

## 6.9 蓝牙接口及驱动程序

在DMA-6410XP实验平台上配置有蓝牙模块。采用Visual Studio 2005编写蓝牙驱动程序，在Windows Embedded CE 6.0下实现对蓝牙模块拨打电话功能，并将结果显示在LCD屏幕上。

### 6.9.1 蓝牙相关知识

蓝牙（Bluetooth）技术是一种短距离无线通信技术。利用“蓝牙”技术，能够有效地简化掌上计算机、笔记本计算机和手持电话等移动通讯终端设备之间的通信，也能够成功地简化以上这些设备与Internet之间的通信，从而使这些现代通讯设备与Internet之间的数据传输变得更加迅速、高效，为无线通信拓宽道路。蓝牙采用分布式网络结构，以及快跳频和短包技术，支持点对点及点对多点通信，工作在全球通用的2.4GHz ISM（即工业、科学、医学）频段，其数据传输速率为1Mbit/s，采用分时双工传输方案实现全双工传输。

S3C6410的蓝牙模块接口电路如图6-56所示。

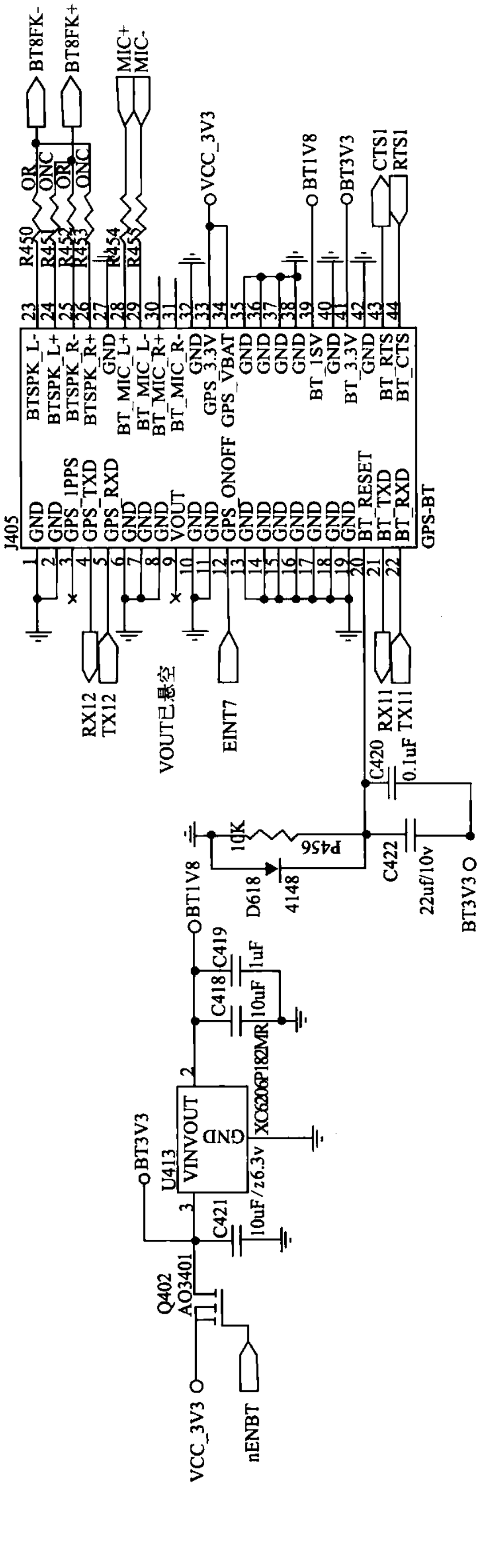



图6-56　S3C6410的蓝牙模块接口电路

## 6.9.2 蓝牙模块的AT命令

以下介绍的所有命令只适用于DMA-6410XP实验平台所搭配的蓝牙模块。

在发送命令时，每条命令应该以“\r\n”结尾。可以使用范例程序中的ATTest.exe来测试以下命令，观察它们的执行效果和返回信息。ATTest.exe是由UART程序修改而来的，其中修改了发送按钮和接收线程中的3行程序代码。

（1）测试命令

命令格式：AT

命令描述：该命令可以测试蓝牙模块是否已经正常工作。如果模块进入了睡眠模式，那么该命令可以唤醒它。

命令返回：正常情况下返回OK。

（2）睡眠时间配置

命令格式：AT+TTDS=seconds

命令描述：该命令用于设置模块的睡眠时间。在默认情况下，该时间设置为0，表示不睡眠。Seconds表示希望在多少秒之后睡眠。

命令返回：当命令执行成功后将返回：TTDS=Seconds，这里的Seconds等于设置的Seconds。

（3）恢复默认配置

命令格式：AT+DFLT

命令描述：该命令用于恢复默认设置。如果当前有连接，那么使用该命令后，将断开所有连接。

命令返回：正常情况下返回OK。

（4）获取蓝牙名称

命令格式：AT+NAME

命令描述：该命令用于获取当前模块的蓝牙名称。

命令返回：NAME=“BlueAudio”

（5）PIN码设置

命令格式：AT+PIN

命令描述：该命令用于获取当前的PIN码。

命令返回：PIN=“pin_code”。Pin_code为当前的PIN码。

命令格式：AT+PIN=“pin_code”

命令描述：该命令用于设置PIN码，pin_code为要设置的PIN码。

命令返回：PIN=“pin_code”。Pin_code为设置的PIN码。

（6）搜寻命令

命令格式：AT+INQ=1

命令描述：开始搜寻周围的设备，该命令在搜寻时间到（61.44秒）或搜寻到8个设备时返回。

命令返回：该命令的返回格式为：INQ=index，Bluetooth_address，name，最后以INQ=E结束。例如：INQ=1，“1e，ad，f5196”，“xiaozhu”。

（7）HSP/HFP连接

命令格式：AT+LH+Iindex

命令描述：与搜寻到的设备进行HSP/HFP连接，连接成功后可以进行有关电话的控制，如拨打接听电话、获取电话簿等操作。index为搜寻返回的index，如使用AT+LH+I1，将连接搜寻到的第一个设备。

命令返回：

```
HFP=1
HFP=2
```

表示连接成功。

```
HFP=1
HFP=0
```

表示连接失败。

```
ERR=12
```

表示错误，没有该index。

（8）拨打电话

命令格式：AT+Dnumber

命令描述：在建立了HSP/HFP连接后，可以使用该命令来进行拨打电话的操作。number为电话号码，如：AT+D10086。

命令返回：

```
HFP=4
```

表示电话已经拨出。

（9）电话操作

命令格式：

AT+CA，接听来电

AT+CR，拒绝接听来电

AT+CH，结束当前的通话

命令返回：正常执行返回OK

（10）A2DP连接

命令格式：AT+LA+Iindex

命令描述：该命令和HSP/HFP的连接命令类似，建立模块与手持电话之间的A2DP连接，当建立该连接后，可以进行音乐播放等操作。

命令返回：

```
A2DP=1
A2DP=2
```

表示建立A2DP连接。

```
A2DP=1
A2DP=0
```

表示建立A2DP连接失败。

```
ERR=12
```

表示错误，没有该index。

（11）音乐播放控制

命令格式：

AT+STP，停止播放

AT+P/P，播放/暂停

AT+FWD，播放前一首

AT+BWD，播放下一首

命令返回：正常情况下返回OK

## 6.9.3 消息

这里所讲的消息是蓝牙模块主动发送给应用的信息。

（1）Ring

当有电话打入时，模块会不断地发送该信息给应用程序。可用于检测来电。

（2）CLID<number>

当有电话打入时，模块会不断地发送该信息给应用程序。number表示来电号码。

## 6.9.4 蓝牙驱动程序设计

本节介绍的蓝牙软件的实现和电话软件有许多类似之处，步骤如下。

### 1. 建立项目并实现软件界面

新建一个名为BlueTooth的项目，然后实现如图6-57所示的界面。

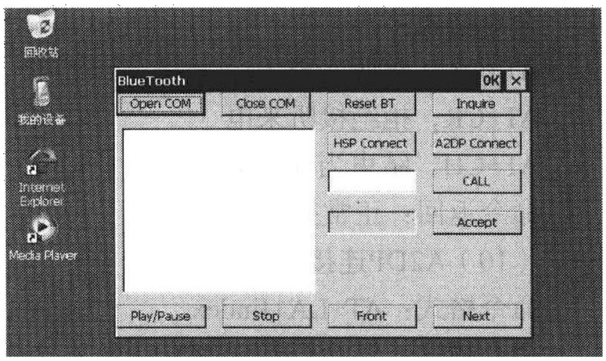



图6-57 BlueTooth软件界面

BlueTooth软件界面左边的白色方框是一个List Box控件，在它的属性中需要设置“Sort”为“False”，不允许它对控件中的字符串排序。

“Open COM”按钮用于打开串行端口，“Close COM”按钮用于关闭串行端口，“Reset BT”重置蓝牙模块，“Inquire”按钮用于搜寻设备，“HSP Connect”按钮用于进行HSP/HFP连接，“A2DP Connect”按钮用于进行A2DP连接，“CALL”按钮用于拨打电话操作，“Accept”按钮用于接听来电，“Play/Pause”按钮用于播放/暂停音乐，“Stop”按钮用于停止播放音乐，“Front”按钮用于播放前一首歌曲，“Next”按钮用于播放下一首歌曲。

### 2. 实现串行端口操作函数

对串行端口进行读/写操作，可以参照电话软件，先实现有关串行端口操作的一些函数，包括读串行端口线程函数ReadThread、写串行端口函数WritePort、信息处理函数ParseData（这个函数先不实现程序代码）、数据接收完成检验函数CheckData、Open COM按钮函数和Close COM按钮函数。

```
//接收串口数据线程
UINT ReadThread(LPVOID pParam)
{
   CBlueToothDlg *Pthis = (CBlueToothDlg*)pParam ;     /*转换参数为CUartDlg类型,以便
                                                      将来可以使用它的成员变量和函数等*/
   if(Pthis == NULL)       //检查参数是否有效
          return 0;
   BYTE    byData;
   DWORD    dwBytes;
   BYTE     byReceiveBuf[1024];   //缓冲区的大小
   int      nBytes= 0;
   DWORD   evtMask;

   //设置字符监听信号
   SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
   //只要m_hcom有效,就一直循环
   while(Pthis->m_hcom != NULL && Pthis->m_hcom != INVALID_HANDLE_VALUE)
   {
          //等待收到串口信号
          if(WaitCommEvent(Pthis->m_hcom, &evtMask,0))
          {
                 //设置字符监听信号
                 SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
                 //清空接收数组
                 memset(byReceiveBuf,0,1024);
                 nBytes = 0;
                 do
                 {
                        //读字符
                        if(ReadFile(Pthis->m_hcom, &byData,1, &dwBytes,0))
                        {
                               if(dwBytes == 1)
                               {
                                      byReceiveBuf[nBytes++] = byData;
                                      if(nBytes == 1024)
                                      {

                                             break;  //接收缓冲区已满
                                      }
                               }
                        }
                 } while(dwBytes == 1);
                 byReceiveBuf[nBytes] = '\0';
                 if(nBytes == 0)   //没接收到数据
                 {
```

```
                    continue;      //继续循环
                }
                TCHAR   szRecv[1024];
                mbstowcs(&szRecv[0], (char*)byReceiveBuf,1024);
                RETAILMSG(1, (TEXT("<%s"),szRecv));
                //开始对收到的字符串进行处理
                Pthis->ParseData((char*)byReceiveBuf);
            }
        }
    return 0;
}


BOOL CBlueToothDlg::WritePort(const char *str)
{
    DWORD dwNumBytesWritten;
    DWORD dwHaveNumWritten =0 ; //已经写入多少
    DWORD bufLen = strlen(str);
    //判断串口是否有效
    if(m_hcom == NULL || m_hcom == INVALID_HANDLE_VALUE)
    {
            return FALSE;
    }
    do
    {
            if(WriteFile(m_hcom,                       //串口句柄
                    str+dwHaveNumWritten,              //被写数据缓冲区
                    bufLen - dwHaveNumWritten,         //被写数据缓冲区大小
                    &dwNumBytesWritten,                //函数执行成功后,返回实际向串口写的个数
                    NULL))                             //此处必须设置NULL
            {
                    dwHaveNumWritten = dwHaveNumWritten + dwNumBytesWritten;
                    //写入完成
                    if(dwHaveNumWritten == bufLen)
                    {
                            break;
                    }
                    Sleep(10);
            }
            else
            {
                    return FALSE;
            }
    }while(TRUE);
    return TRUE;
}


void CBlueToothDlg::ParseData(const char *data)
{
    //如果没有收到完成的信息,不处理,等到收到后再处理
    if(!CheckData(data))
```

```
        return;

    if(ParseChar(m_data, "INQ=E"))
    {
        memset(m_data,0,sizeof(m_data));
        return;
    }
    //处理搜索的返回信息，并把设备名称添加到List列表中
    if(ParseChar(m_data, "INQ="))
    {
        CString name;
        GetName(m_data,name);
        ((CListBox*)GetDlgItem(IDC_LIST_INQUIRE))->AddString(name);
        memset(m_data,0,sizeof(m_data));
        return;
    }
    //处理来电号码信息，把号码显示在ACCEPT文本框中
    if(ParseChar(m_data, "CLID"))
    {
        char number[32];
        //移动指针到号码开始位置
        char *pfind = strstr(m_data, "D");
        pfind++;
        int i=0;
        //复制号码
        while(i<31 && *pfind)
        {
            if(*pfind == '\r')
            {
                break;
            }
            number[i] = *pfind;
            i++;
            pfind++;
        }
        number[i] = '\0';
        //显示号码
        CString nb;
        nb = number;
        SetDlgItemText(IDC_EDIT_ACCEPT,nb);

        memset(m_data,0,sizeof(m_data));
        return;
    }

    memset(m_data,0,sizeof(m_data));
}


//确认信息是否完全收到，若完全收到，则返回TRUE，否则FALSE
BOOL CBlueToothDlg::CheckData(const char *data)
{
```

```
        //判断该数据的最后两位是否是\r\n
        if(data[strlen(data)-2] == '\r' && data[strlen(data)-1] == '\n')
        {
                if(m_data[0] == 0)//这条信息是第一次收到,且全部收到
                {
                        strcpy(m_data,data);
                        return TRUE;
                }
                else    //这条信息不是第一次收到,但是全部收完
                {
                        strcat(m_data,data);
                        return TRUE;
                }
        }
        else    //没有收到一条信息的结尾
        {
                if(m_data[0] == 0)//这条信息是第一次收到,没有全部收到
                {
                        strcpy(m_data,data);
                        return FALSE;
                }
                else    //这条信息不是第一次收到,并且没有全部收完
                {
                        strcat(m_data,data);
                        return FALSE;
                }
        }
}


void CBlueToothDlg::OnBnClickedButtonOpencom()
{
        //打开串口
        m_hcom = CreateFile(_T("COM1:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,0,NULL);
        //检测是否打开串口成功
        if(m_hcom != INVALID_HANDLE_VALUE && m_hcom != NULL)
        {
                DCB  dcb;
                dcb.DCBlength = sizeof(DCB);
                //配置串口参数
                GetCommState(m_hcom, &dcb);
                dcb.BaudRate = CBR_115200 ;    //波特率
                dcb.ByteSize = 8;              //数据位
                dcb.Parity = NOPARITY;         //奇偶校验
                dcb.StopBits = ONESTOPBIT;     //停止位

                if(!SetCommState(m_hcom, &dcb))
                {
                        //配置串口失败
                        MessageBox(_T("Set COM port fail!"));
                        return ;
                }
```

```
            //配置超时值
            COMMTIMEOUTS  cto;
            GetCommTimeouts(m_hcom, &cto);
            cto.ReadIntervalTimeout = MAXDWORD;
            cto.ReadTotalTimeoutMultiplier = 0;
            cto.ReadTotalTimeoutConstant = 0;
            cto.WriteTotalTimeoutMultiplier = 10;
            cto.WriteTotalTimeoutConstant = 1000;
            if(!SetCommTimeouts(m_hcom, &cto))
            {
                //不能设置超时参数
                MessageBox(_T("Set COM timeout fail!"));
                return ;
            }
        }
        else
        {
            MessageBox(_T("Open COM fail!"));
            return ;
        }
        //串口打开有效,启动读串口线程
        DWORD m_dwThreadID;
        HANDLE m_hThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)ReadThread,this,0, &m_dwThreadID);
        CloseHandle(m_hThread);
    }

    void CBlueToothDlg::OnBnClickedButtonClosecom()
    {
        if(m_hcom)
        {
            CloseHandle(m_hcom);
            m_hcom = NULL;
        }
    }
```

### 3. 实现“Reset BT”按钮

在“Reset BT”按钮中实现发送AT+DFLT命令，重新启动蓝牙模块。模块重新启动时会断开当前的所有连接。

```
    void CBlueToothDlg::OnBnClickedButtonResetbt()
    {
        //重置蓝牙模块,会断开当前的所有连接,但不会修改PIN码
        WritePort("AT+DFLT\r\n");
    }
```

### 4. 实现“Inquire”按钮

在“Inquire”按钮中，首先把List列表中的内容清空，因为接下来要把搜寻到的信息显示在该列表中。然后，发送AT+INQ=1命令，搜寻周围的设备。

```
    void CBlueToothDlg::OnBnClickedButtonInquire()
    {
```

```
    //在每次搜索之前先删除List列表中所有的字符串
    int len = ((CListBox*)GetDlgItem(IDC_LIST_INQUIRE))->GetCount();
    for(int i=0; i<len; i++)
    {
            ((CListBox*)GetDlgItem(IDC_LIST_INQUIRE))->DeleteString(0);
    }

    //搜索命令,当搜索时间(61.44秒)到或寻找到个设备的时候返回
    WritePort("AT+INQ=1\r\n");
}
```

### 5. 处理搜寻的返回信息

在发送了搜寻命令后，当搜寻的时间（61.44秒）到或搜寻到8个设备的时候，模块会返回搜寻结果信息。把这些信息中的有关设备名称的数据提取出来，然后显示在List列表中。

```
void CBlueToothDlg::GetName(const char *data,CString &name)
{
    char n[32];
    memset(n,0,32);
    const char *p = data;
    //把指针移动到名字的开始位置
    for(int i=0; i<3; i++)
    {
            p = strchr(p, '"');
            if(NULL == p)
                    return;
            p++;
    }
    //把data数据中的名字字符复制到n数组中
    int i=0;
    while(*p != '"' && i<31)
    {
            n[i] = *p;
            p++;
            i++;
    }
    n[i] = 0;
    /*把n数组中的数据转换成String型,这里的n数组中的数据是UTF编码格式,要把UTF编码转换成Unicode编码*/
    UTF2Uni(n,name);
}
```

使用GetName函数实现从信息中获取名称。返回信息中的设备名称的编码是UTF格式，需要把它们转换成Unicode编码格式。可以使用UTF2Uni函数来实现这个功能。

### 6. 实现“HSP Connect”按钮

在软件操作时，用户先选择List列表中的相应设备的名称，然后再单击这个按钮，实现相应的连接。

如果要实现这样的功能，就需要知道在List列表中选中了哪个设备，使用GetCurSel函数可以获取到该选择的index，但要注意的是，这里的index是从0开始的，要对应到蓝牙搜寻结果的index，需要加1。

有了这个index后，组合成一个类似AT+LH+I1这样的命令来连接设备。

```
void CBlueToothDlg::OnBnClickedButtonHspconnect()
{
    int index=0;
    //获取当前在List中选中的字符串的index,注意该index是从0开始的
    index = ((CListBox*)GetDlgItem(IDC_LIST_INQUIRE))->GetCurSel();
    if(index == -1)         //没有选中任何字符串
    {
            return;
    }
    index++;
    //构造连接的命令
    CString cmd;
    cmd.Format(_T("AT+LH+I%d\r\n"),index);
    //字符转换
    char s[64];
    wcstombs(s,cmd,64);
    //发送命令
    WritePort(s);
}
```

### 7. 实现“CALL”按钮

当建立HSP/HFP连接后，可以进行有关电话的操作，在该按钮中先获取CALL文本框中的号码，然后组合成类似AT+D10086这样的命令，发送该命令以实现拨打电话的功能。

```
void CBlueToothDlg::OnBnClickedButtonCall()
{
    //获取文本框中的号码
    CString number;
    GetDlgItemText(IDC_EDIT_CALL,number);
    //组合命名
    CString cmd;
    cmd = _T("AT+D") + number + _T("\r\n");
    //字符转换
    char s[64];
    wcstombs(s,cmd,64);
    //发送命令
    WritePort(s);
}
```

### 8. 实现来电显示的功能

当有电话打入时，将收到CLID开头的信息，需要在ParseData函数中处理该信息，然后把信息中有关电话号码的信息提取出来，显示在Accept文本框中。

```
void CBlueToothDlg::OnBnClickedButtonAccept()
{
    //接听电话
    WritePort("AT+CA\r\n");
}
```

### 9. 实现“A2DP Connect”按钮

在“A2DP Connect”按钮中实现有关A2DP连接的程序代码。该函数的实现和“HSP Connect”按钮的程序代码类似，同样先获取List列表中选中的index，然后组合成连接命令，发送该命令。

```
void CBlueToothDlg::OnBnClickedButtonA2dpconnect()
{
    int index=0;
    //获取当前在List中选中的字符串的index,注意,该index是从0开始的
    index = ((CListBox*)GetDlgItem(IDC_LIST_INQUIRE))->GetCurSel();
    if(index == -1)         //没有选中任何字符串
    {
            return;
    }
    index++;
    //构造连接的命令
    CString cmd;
    cmd.Format(_T("AT+LA+I%d\r\n"),index);
    //字符转换
    char s[64];
    wcstombs(s,cmd,64);
    //发送命令
    WritePort(s);
}
```

### 10. 实现播放控制的4个按钮

在播放控制的4个按钮中，分别使用AT+P/P、AT+STP、AT+FWD、AT+BWD命令实现相应的控制。

# 第三部分　项目篇


- 第7章　GPS与Google Earth定位系统
- 第8章　鲜奶配送系统
- 第9章　无线点餐系统

# 第7章　GPS与Google Earth定位系统

## 7.1　GPS相关知识

全球定位系统（Global Positioning System，GPS）是美国从20世纪70年代开始研制，历时20年，耗资200亿美元，于1994年全面建成，具有在海、陆、空皆可进行全方位实时三维导航与定位能力的新一代卫星导航与定位系统。经近10年我国测绘等部门的使用说明，GPS以全天候、高精度、自动化、高效益等显著特点，赢得广大测绘工作者的信赖，并成功地应用于大地测量、工程测量、航空摄影测量、运载工具导航和管制、地壳运动监测、工程变形监测、资源勘察、地球动力学等多种学科，从而给测绘领域带来一场深刻的技术革命。

随着全球定位系统的不断改良，软硬件技术的不断改进，应用领域正在不断地开拓，目前已遍及国民经济各种部门，并开始逐步深入人们的日常生活。

GPS系统包括三大部分：空间部分——GPS卫星星座；地面控制部分——地面监控系统；用户设备部分——GPS信号接收机。

### 7.1.1　卫星星座

由21颗工作卫星和3颗在轨备用卫星组成GPS卫星星座，记作（21+3）GPS星座。24颗卫星均匀分布在6个轨道平面内，轨道倾角为55°，各个轨道平面之间夹角为60°，即轨道的升交点赤经各相差60°。每个轨道平面内各颗卫星之间的升交角距相差90°，同一轨道平面上的卫星比西边相邻轨道平面上的相应卫星超前30°。

在两万公里高空的GPS卫星，当地球对恒星来说自转一周时，它们绕地球运行两周，即绕地球一周的时间为12恒星时。这样，对于地面观测者来说，每天将提前4分钟见到同一颗GPS卫星。位于地平线以上的卫星颗数，随着时间和地点的不同而不同，最少可见到4颗，最多可见到11颗。在用GPS信号导航定位时，为了结算观测站的三维坐标，必须观测4颗GPS卫星，称为定位星座。这4颗卫星在观测过程中的几何位置分布对定位精度有一定的影响。对于某地某时，甚至不能测得精确的点位坐标，这种时间段叫做“间隙段”。但这种间隙段是很短暂的，并不影响全球绝大多数地方的全天候、高精度、连续实时的导航定位测量。GPS工作卫星的编号和试验卫星基本相同。

### 7.1.2 地面监控系统

对于导航定位来说，GPS卫星是一动态已知点。卫星的位置是依据卫星发射的星历——描述卫星运动及其轨道的参数算得。每颗GPS卫星所播发的星历，是由地面监控系统提供的。卫星上的各种设备是否正常工作，以及卫星是否一直沿着预定轨道运行，都要由地面设备进行监测和控制。地面监控系统另一重要作用是保持各颗卫星处于同一时间标准——GPS时间系统。这就需要地面站监测各颗卫星的时间，求出钟差。然后由地面注入站发给卫星，卫星再由导航电文发给用户设备。GPS工作卫星的地面监控系统包括一个主控站、三个信息注入站和五个监测站。

### 7.1.3 信号接收机

GPS信号接收机的任务是：能够捕获到按一定卫星高度截止角所选择的待测卫星的信号，并追踪这些卫星的运行，对所接收到的GPS信号进行变换、放大和处理，以便测量出GPS信号从卫星到接收机天线的传输时间，解译出GPS卫星所发送的导航电文，实时地计算出观测站的三维位置，甚至三维速度和时间。

静态定位中，GPS接收机在捕获和追踪GPS卫星的过程中固定不变，接收机高精度地测量GPS信号的传输时间，利用GPS卫星在轨道的已知位置，解析算出接收机天线所在位置的三维坐标。而动态定位则是用GPS接收机测定一个运动物体的运行轨迹。GPS信号接收机位于的运动物体叫做载体（如航行中的船舰，空中的飞机，行走的车辆等）。载体上的GPS接收机天线在追踪GPS卫星的过程中相对地球而运动，接收机用GPS信号实时地测得运动载体的状态参数（瞬间三维位置和三维速度）。

接收机硬件和软件，以及GPS数据的后置处理软件包，构成了完整的GPS用户设备。GPS接收机的结构分为天线单元和接收单元两大部分。对于测地型接收机来说，两个单元一般分成两个独立的部件，观测时将天线单元安置在观测站上，接收单元置于观测站附近的适当地方，用电缆线将两者连接成一个整机。也有的将天线单元和接收单元制作成一个整体，观测时将其安置在观测站点上。

GPS接收机一般用蓄电池做电源。同时采用机内、机外两种直流电源。设置机内电池的目的，在于更换外电池时不中断连续观测。在用机外电池的过程中，机内电池自动充电。关机后，机内电池为RAM内存供电，以防止丢失数据。

近几年，国内引进了许多种类型的GPS测地型接收机。各种类型的GPS测地型接收机用于精密相对定位时，其双频接收机精度可达5 mm+1 PPM.D，单频接收机在一定距离内精度可达10 mm+2 PPM.D。用于差分定位其精度可达亚米级至厘米级。

随着嵌入式技术和通信技术的迅猛发展，GPS导航已经逐步融入人们日常的生活、工作中。嵌入式电脑、PDA、手机都可以非常方便地接入GPS设备，实现导航功能。目前GPS设备大体有两种常用方式接入嵌入式电脑，一种是Rs232串口，另外一种是蓝牙。对于Windows CE中的应用程序，都可通过串口服务来访问GPS信息。而在嵌入式电脑、PDA或者智能手机中，

只需要将蓝牙连接方式设置成串口服务方式，指定虚拟的串口号，就可以通过串口编程进行通信了。本实例通过实现GPS定位，并结合Google Earth制作一个Google Earth定位系统。

## 7.2 搭建环境

程序在Visual Studio 2005开发环境中编写与编译，使用C++与MFC作为开发的语言环境。在程序的建立过程中需要下载Google Earth软件，利用Google Earth COM API类库实现地图定位。

### 7.2.1 解析GPS定位信息

目前许多的GPS厂商都遵循NMEA0183协议。NMEA0183是一种航海、海运方面有关于数字信号传递的标准，此标准定义了电子信号所需要的传输协议，传输数据时间，并且指明了信息格式需要一个4800bit/s的串行数据接口。

NMEA0183协议中包含了很多有关GPS信息的封包格式，结合实验平台上的GPS模块，这里只对GPRMC格式的封包格式进行解析：

推荐的最简定位信息（GPRMC）：$GPRMC，121212.456，A，3232.1234，N，12121.3322，W，0.15，305.12，121299，*22$GPRMC，<1>，<2>，<3>，<4>，<5>，<6>，<7>，<8>，<9>，<10>，*hh。

<1>UTC时间（定位点）：hhmmss.sss格式。

<2>状态：A=定位（有效定位），V=导航（无效定位）。

<3>纬度：ddmm.mmm格式，当位数不够时，将以0补齐并传送。

<4>纬度方向：N或S。

<5>经度：dddmm.mmmm格式，当位数不够时，将以0补齐并传送。

<6>经度方向：E或W。

<7>对地航速（单位：海里／小时，又称“节”）。

<8>对地航向（二维方向指向，相当于二维罗盘，单位：度）。

<9>当前UTC日期（定位点）：ddmmyy格式。

<10>磁偏角（单位：度）。

### 7.2.2 下载并安装Google Earth

到Google官方网站（http://earth.google.com）下载并安装最新的Google Earth客户端。先下载的是一个约500KB的可执行文件，执行该文件，将自动开始下载Google Earth客户端并完成安装。使用这种方法进行安装，不要下载使用一些绿色免安装版。使用这些版本可能导致后面编写的应用不能成功地调用Google Earth。

COM为组件技术，API为应用程序接口。Google公司为用户提供了Google Earth COM API类别库，其意义在于用户可以在各种外部程序中调用Google Earth的功能。Google Earth COM

API是一个包括11个类别的类别库，这些类别都实现了IDispatch接口。IDispatch接口和COM中的自动化技术相关。

由于本设计是创建在MFC的开发环境之中，这里介绍一下MFC对自动化技术的支持。MFC为支持该技术提供了一个类别：COleDispatchDriver类别。在MFC中添加的Google Earth COM API类别库中的类别，实际上都是COleDispatchDriver类别的衍生类别。因此，用户在使用Google Earth COM API类别库时，都有可能用到COleDispatchDriver类别的属性和函数。

COleDispatchDriver类别的内容比较复杂，这里简单介绍几个必要的函数。CreateDispatch()成员函数通过组建的CLSID值来创建组建对象，并返回该对象的指针。AttachDispatch()成员函数可以把当前COleDispatchDriver类别的对象和其他自动化对象联系起来。DetachDispatch()函数和AttachDispatch()函数的功能相反。ReleaseDispatch()函数用来释放所创建的对象。

用户通过COleDispatchDriver类别的函数创建了Google Earth的对象，进而可以使用Google Earth COM API类别库。该类别库的11个类别中，CApplicationGE类别最为重要。Google Earth COM API类别库中的其他类别的使用，都会涉及CApplicationGE类别中的属性或函数。这些类别的简单介绍，如表7-1所示。

表7-1 类别库

| 名 称 | 含 义 |
|---|---|
| CApplicationGE | 入口类别，通过该类别，用户进一步调用其他类别 |
| CCameraInfoGE | 相机类别，通过该类别，用户可以调整观看当前视图的方式 |
| CFeatureGE | 要素类别，通过该类别，用户可以控制要素的属性 |
| CFeatureCollectionGE | 要素集合类别，通过该类别，用户进一步获取要素 |
| CPointOnTerrainGE | 地理坐标点类别，通过该类别，用户获取屏幕点的地理坐标 |
| CViewExtentsGE | 窗口类别，通过该类别，用户可以控制当前的窗口 |
| CSearchControllerGE | Search面板类别，通过该类别，用户可以完成相应的搜索功能 |
| CTourControllerGE | Tour面板类别，通过该类别，用户可以动态播放当前的要素 |
| CAnimationControllerGE | Animation面板类别，通过该类别，用户可以动态播放当前的时间要素 |
| CTimeGE | 时间类别，通过该类别，用户可以获取和设置要素的时间属性 |
| CTimeIntervalGE | 时间间隔类别，通过该类别，用户可以获取要素的时间间隔属性 |

## 7.3 GPS定位程序

下面建立Windows CE端的GPS定位程序来获取用户当前位置的经纬度坐标。可以通过串口通信的方式来获取GPS模块中有关定位的信息。针对该实验平台上配套的GPS模块，使用的串行端口是COM3，波特率为4800bit/s。

1）建立名为GPS的MFC项目并打开。

2）修改编译生成的可执行文件名。选择“Project”→“Properties”命令，在弹出的对话框的左侧选择“Configuration Properties”→“Linker”→“General”，修改右侧的“Output File”选项为$（OutDir）\Gps.exe。同时修改“Debug”和“Release”两种模式的选项。如图7-1所示。

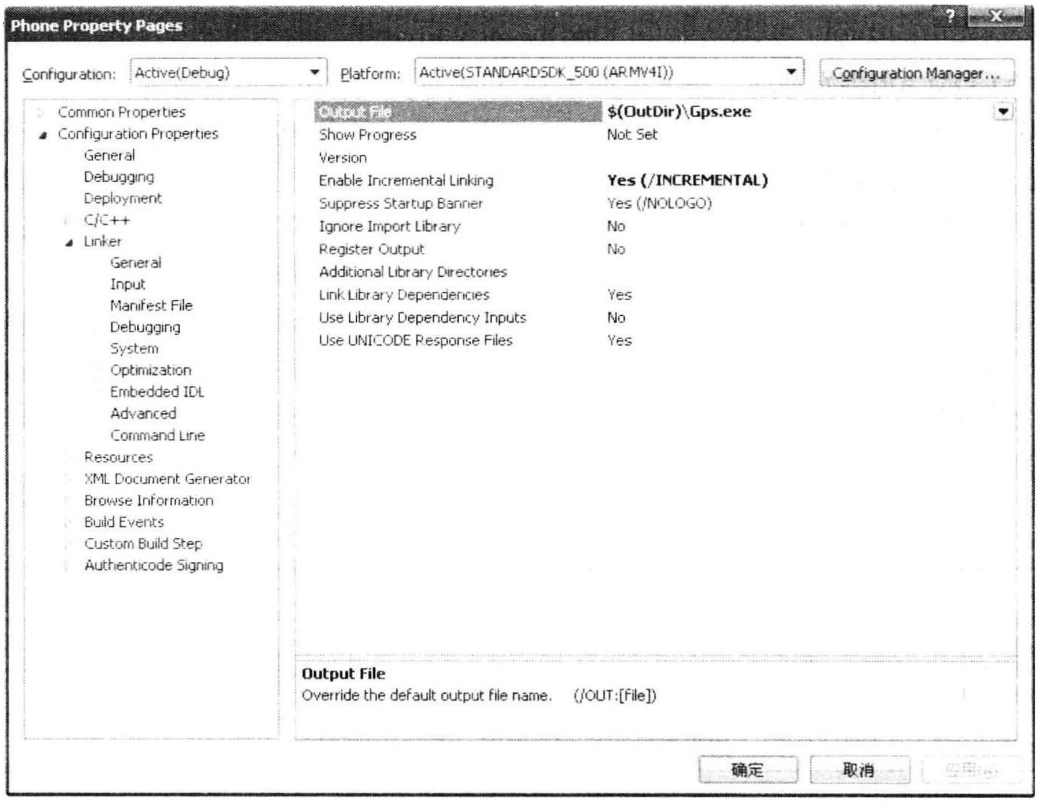



图7-1 修改编译生成的可执行文件名

3）建立程序界面。

设置对话框的“Caption”属性为“Gps”。添加三个STATIC静态文本控件，设置它们的ID属性分别为IDC_STATIC_STATE（状态），IDC_STATIC_LONGITUDE（经度），IDC_STATIC_LATITUDE（纬度），分别用于显示定位信息的状态、经度值和纬度值。并添加OPEN按钮与CLOSE按钮，其ID属性分别为IDC_BUTTON_OPEN、IDC_BUTTON_CLOSE，如图7-2所示。

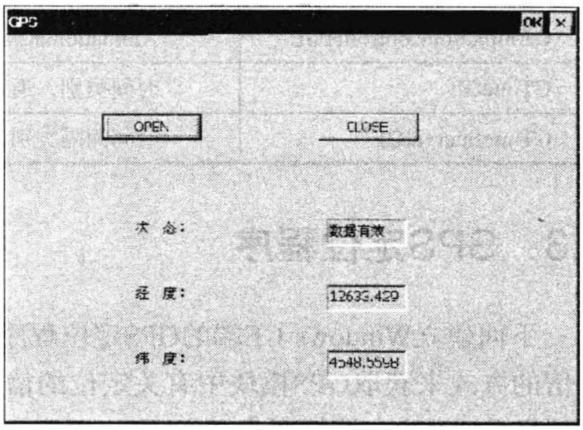



图7-2 建立程序界面

4）设计OPEN按钮功能。Windows CE与GPS模块通过串口通信方式进行各种参数的读写操作。因此，程序需要通过OPEN按钮来打开串口，并进行一些参数的设置。

OPEN打开串口代码如下：

```
void CGpsDlg::OnBnClickedButtonOpen()
{
    //打开串口
    m_hcom = CreateFile(_T("COM3:"),GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,0,NULL);
    //检测是否打开串口成功
    if(m_hcom != INVALID_HANDLE_VALUE && m_hcom != NULL)
    {
            DCB  dcb;
            dcb.DCBlength = sizeof(DCB);
            //配置串口参数
            GetCommState(m_hcom, &dcb);
            dcb.BaudRate = CBR_4800 ;      //波特率
            dcb.ByteSize = 8;              //数据位
            dcb.Parity = NOPARITY;         //奇偶校验
            dcb.StopBits = ONESTOPBIT;     //停止位
            if(!SetCommState(m_hcom, &dcb))
            {
                    //配置串口失败
                    MessageBox(_T("Set COM port fail!"));
                    return ;
            }
            //配置超时值
            COMMTIMEOUTS  cto;
            GetCommTimeouts(m_hcom, &cto);
            cto.ReadIntervalTimeout = MAXDWORD;
            cto.ReadTotalTimeoutMultiplier = 0;
            cto.ReadTotalTimeoutConstant = 0;
            cto.WriteTotalTimeoutMultiplier = 10;
            cto.WriteTotalTimeoutConstant = 1000;
            if(!SetCommTimeouts(m_hcom, &cto))
            {
                    //不能设置超时参数
                    MessageBox(_T("Set COM timeout fail!"));
                    return ;
            }
    }
    else
    {
            MessageBox(_T("Open COM fail!"));
            return ;
    }
    //串口打开有效,启动读串口线程
    DWORD m_dwThreadID;
    HANDLE m_hThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)ReadThread,this,0, &m_dwThreadID);
    CloseHandle(m_hThread);

    //发送ATE0,关闭“回声”
    //注意所有的命令使用\r\n结尾,表示一条命令结束
    WritePort("ATE0\r\n");
    //延时秒,这个很重要,不要连续地发送多条命令给模块,由于模块处理前一条命令
    //需要时间,所以可能导致不能处理紧接着的命令
    //延时是为了让模块有时间处理了上一条命令,再发送下一条命令
    Sleep(1000);    //该函数的参数单位是毫秒
```

```
    //发送AT+CLIP=1,显示来电信息
    //设置了这个命令才可以在串口收到的消息中,解析出来电号码
    WritePort("AT+CLIP=1\r\n");
```

其中，读取线程的ReadThread函数定义如下：

```
UINT ReadThread(LPVOID pParam)
{
    CGpsDlg *Pthis = (CGpsDlg*)pParam ;   //转换参数为CUartDlg类型,以便于后面可以
    //使用它的成员变量和函数等
    if(Pthis == NULL)       //检查参数是否有效
           return 0;
    BYTE byData;
    DWORD dwBytes;
    BYTE byReceiveBuf[1024];  //缓冲区的大小
    int nBytes= 0;
    DWORD  evtMask;

    //设置字符监听信号
    SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
    //只要m_hcom有效,就一直循环
    while(Pthis->m_hcom != NULL && Pthis->m_hcom != INVALID_HANDLE_VALUE)
    {
           //等待收到串口信号
           if(WaitCommEvent(Pthis->m_hcom, &evtMask,0))
           {
                  //设置字符监听信号
                  SetCommMask(Pthis->m_hcom,EV_RXCHAR | EV_CTS | EV_DSR);
                  //清空接收数组
                  memset(byReceiveBuf,0,1024);
                  nBytes = 0;
                  do
                  {
                         //读字符
                         if(ReadFile(Pthis->m_hcom, &byData,1, &dwBytes,0))
                         {
                                if(dwBytes == 1)
                                {
                                       byReceiveBuf[nBytes++] = byData;
                                       if(nBytes == 1024)
                                       {

                                              break;  //接收缓冲区已满
                                       }
                                }
                         }
                  } while(dwBytes == 1);
                  byReceiveBuf[nBytes] = '\0';
                  if(nBytes == 0)  //没接到数据
                  {
                         continue;       //继续循环
                  }
                  TCHAR   szRecv[1024];
                  mbstowcs(&szRecv[0], (char*)byReceiveBuf,1024);
```

```
                RETAILMSG(1, (TEXT("<%s"),szRecv));
                //开始对收到的字符串进行处理
                Pthis->ParseData((char*)byReceiveBuf);
            }
        }
    return 0;
}
```

WritePort函数用来向串口写入数据，定义如下：

```
BOOL CGpsDlg::WritePort(const char *str)
{
    DWORD dwNumBytesWritten;
    DWORD dwHaveNumWritten =0 ; //已经写入多少
    DWORD bufLen = strlen(str);
    //判断串口是否有效
    if(m_hcom == NULL || m_hcom == INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }
    do
    {
        if(WriteFile(m_hcom,                    //串口句柄
                str+dwHaveNumWritten,           //被写数据缓冲区
                bufLen - dwHaveNumWritten,      //被写数据缓冲区大小
                &dwNumBytesWritten,             //函数执行成功后,返回实际向串口写的个数
                NULL))                          //此处必须设置NULL
        {
                dwHaveNumWritten = dwHaveNumWritten + dwNumBytesWritten;
                //写入完成
                if(dwHaveNumWritten == bufLen)
                {
                        break;
                }
                Sleep(10);
        }
        else
        {
                return FALSE;
        }
    }while(TRUE);
    return TRUE;
}
```

ParseData函数代码如下：

```
void CGpsDlg::ParseData(const char *data)
{
    //如果没有收到完成的信息,不处理,等待收到再处理
    if(!CheckData(data))
        return;

    char c_data[1024];
    strcpy(c_data,m_data);
    memset(m_data,0,1024);
```

```
//判断是否有$GPRMC开头的信息
const char *fdata = strstr(c_data, "$GPRMC");
if(fdata)//如果有该信息
{
        CString str_longitude;
        CString str_latitude;
        //把指针移动到第二位,判断是否是有效的定位信息
        for(int i=0; i<2; i++)
        {
                fdata = strchr(fdata, ',');
                if(fdata == NULL)
                        return;
                fdata++;
        }
        if(fdata[0] != 'A')    //定位无效,返回
        {
                ((CStatic*)GetDlgItem(IDC_STATIC_STATE))->SetWindowTextW(_T("状态: 有效的定位资讯"));
                return;
        }
        ((CStatic*)GetDlgItem(IDC_STATIC_STATE))->SetWindowTextW(_T("状态: 有效的定位资讯"));
        //纬度
        fdata = strchr(fdata, ',');
        if(fdata == NULL)
                return;
        fdata++;
        char lat[12];
        memset(lat,0,12);
        strncpy(lat,fdata,12);
        lat[9] = 0;
        str_latitude = lat;
        //纬度半球N(北半球)或S(南半球)
        fdata = strchr(fdata, ',');
        if(fdata == NULL)
                return;
        fdata++;
        if(fdata[0] == 'S')    //南纬被表示成负值
        {
                str_latitude.Insert(0, '-');
        }
        //经度dddmm.mmmm(度分)格式(前面的也将被传输)
        fdata = strchr(fdata, ',');
        if(fdata == NULL)
                return;
        fdata++;
        memset(lat,0,12);
        strncpy(lat,fdata,12);
        lat[10] = 0;
        str_longitude = lat;
        //经度半球E(东经)或W(西经)
        fdata = strchr(fdata, ',');
        if(fdata == NULL)
                return;
```

```
            fdata++;
            if(fdata[0] == 'W')    //西经被表示成负值
            {
                    str_longitude.Insert(0, '-');
            }
            //显示经纬度值
            ((CStatic*)GetDlgItem(IDC_STATIC_LONGITUDE))->SetWindowTextW(_T("经度: ")+str_longitude);
            ((CStatic*)GetDlgItem(IDC_STATIC_LATITUDE))->SetWindowTextW(_T("纬度: ")+str_latitude);

    }
}
```

ParseData函数执行原理是，先判断收到的信息中是否有存在“$GPRMC”这样的标头，如果有，则判断该条信息是否是有效的定位信息，如果是有效的定位信息，则撷取该信息中的经纬度坐标值，并显示它们。

最后通过CLOSE按钮单击事件CLOSE函数来关闭串口。代码如下：

```
void CGpsDlg::OnBnClicked Button Close()
    {
    if(m_hcom)
    {
            CloseHandle(m_hcom);
            m_hcom = NULL;
    }
}
```

5）执行测试该软件。执行该软件后，先点击“Open”按钮打开串行端口，此时将显示定位信息，如果是有效的定位信息，还会显示出当前点的经纬度值。

## 7.4　GPS与Google Earth定位程序

该程序是在PC端执行的程序，前提是安装新版本的Google Earth软件，并能够调用其API。最终实现的功能是在程序中输入经纬度坐标，程序自动调用Google Earth来指向该坐标的准确地点。

### 7.4.1　Google Earth_Start程序的建立

1）建立一个名为GoogleEarth_Start的MFC项目，注意要建立的是PC端执行程序的项目。打开Visual Studio 2005，选择“File”→“new”→“project”出现如图7-3所示界面，其中Project types选择MFC，右侧选择MFC Application，下方程序名称为GoogleEarth_Start，点击OK进入overview界面。

Overview界面如图7-4所示。点击Next进入程序类型选择，如图7-5所示。其中Application Type选择Dialog Based。Use of MFC选择Use MFC in a static library。语言选择中文（中国）。之后步骤默认即可，因此点击Finish完成GoogleEarth_Start程序的建立。

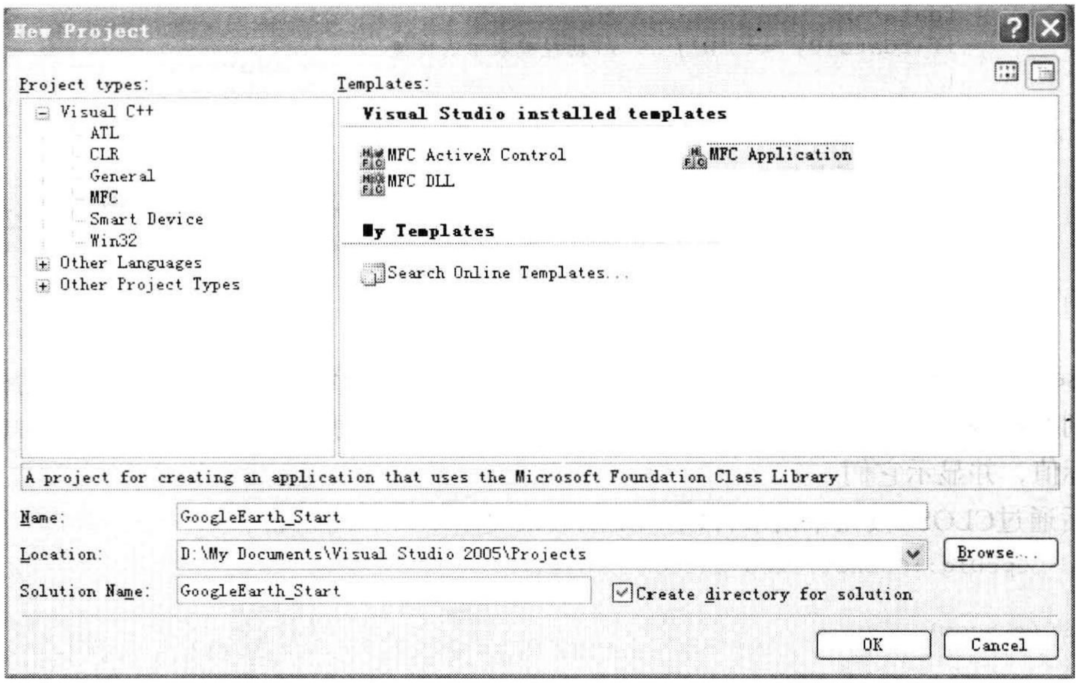



图7-3　建立MFC程序GoogleEarth_Start

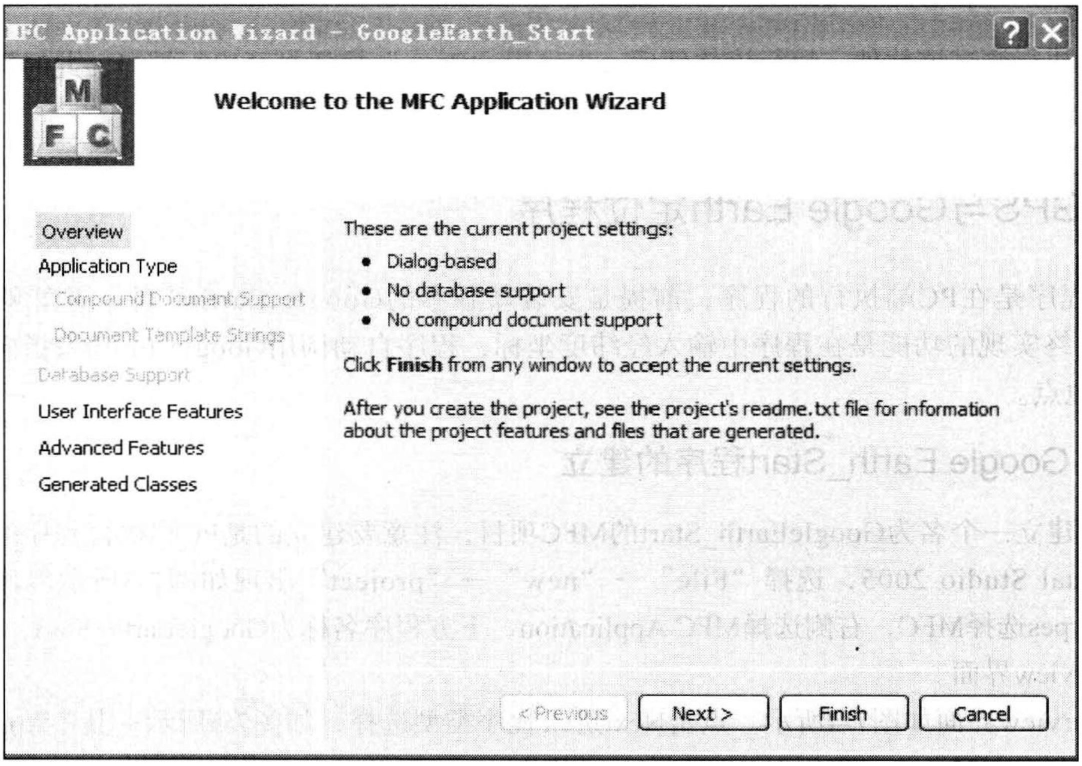



图7-4　GoogleEarth_Start程序的建立overview

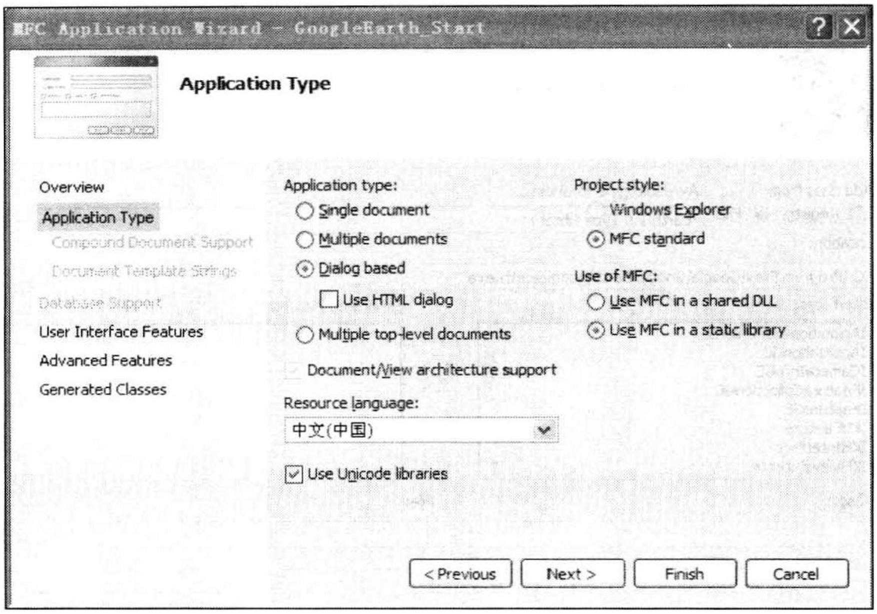



图7-5 GoogleEarth_Start程序的建立（Application Type）

2）添加Google Earth API类别。若之前已经成功安装GoogleEarth软件，则下一步将在GoogleEarth_Start程序中添加GoogleEarth自带的名为CApplicationGE的API类。

在菜单栏依次选择“Project”→“Add Class”，在出现的对话框中，左侧选择“Visual C++”→“MFC”，在右侧选择“MFC Class From TypeLib”。如图7-6所示。点击“Add”按钮，在弹出的对话框中，选择“Add class from”为“File”。然后点击“Location:”项后面的按钮，此时将会弹出一个对话框，选择Google Earth的可执行文件“googleearth.exe”（通常该文件在“C:\Program Files\ Google\Google Earth\googleearth.exe”路径下），选择该文件后将显示如图7-7所示界面。

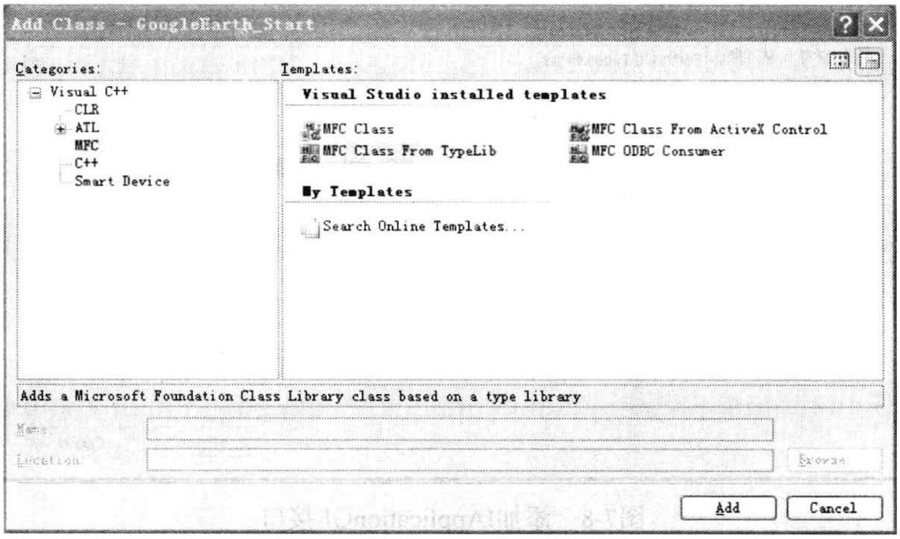



图7-6 添加MFC class From TypeLib

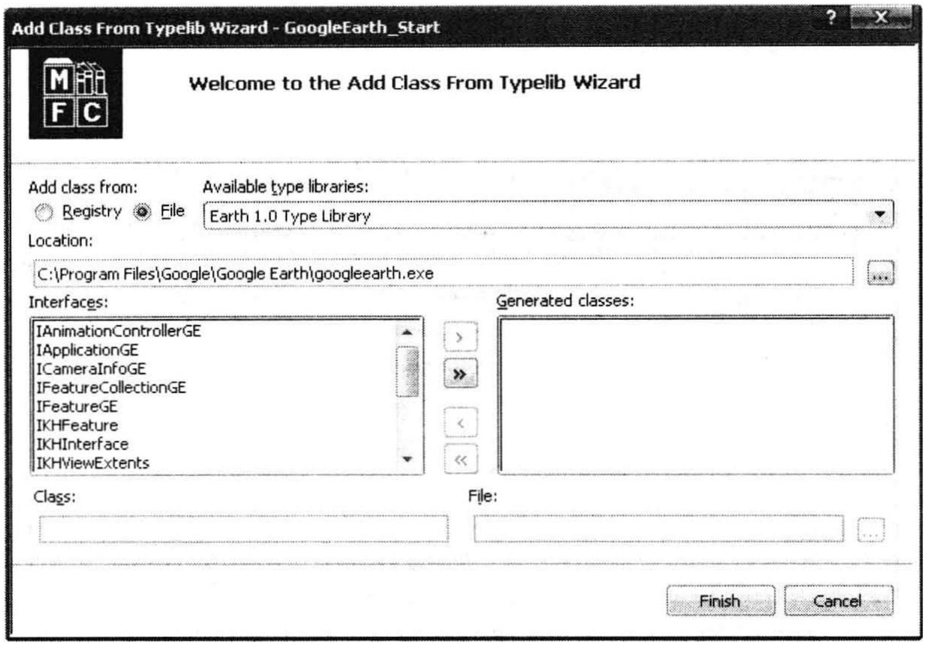



图7-7 添加Google Earth API类别

在Interfaces列表框中列出了15个接口类别，其中有4个以IKH开头的接口是Google Earth以前版本的接口。它们的功能基本上已经被新的接口替代并有所改进。

在左侧列表框中可以通过点击两下或选中使用按钮来添加IApplicationGE接口，如图7-8所示。

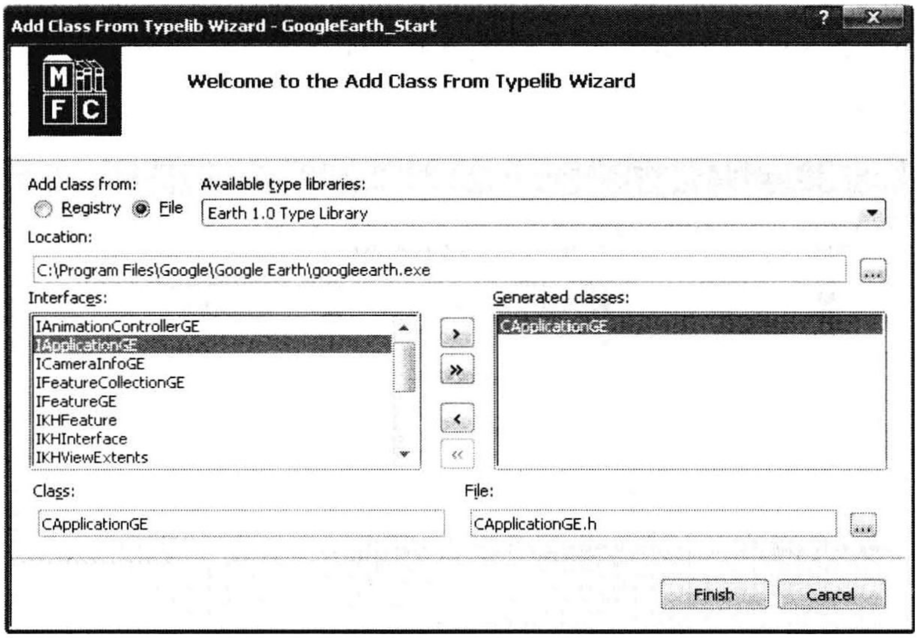



图7-8 添加IApplicationGE接口

点击“Finish”按钮，完成接口类别的添加。此时在“Solution Explorer”窗口中将看到有新添加一个名为“CApplicationGE.h”的文件。

3）设计软件接口。该软件的接口很简单，只需要添加一个启动按钮即可，设置该按钮的“Caption”属性为“StartUp Google Earth”，然后给该按钮添加一个点选事件。

4）添加程序代码。在“GoogleEarth_StartDlg.h”文件中添加对“CApplicationGE.h”文件的引用：

```
#include "CApplicationgGE.h"
```

并且定义CApplicationGE类别的对象：

```
CApplicationGE     m_geApplication;
```

在“GoogleEarth_StartDlg.cpp”中，定义一个静态的CLSID常量，程序代码如下：

```
static const CLSID CLSID_GoogleEarth = {0x8097D7E9,0xDB9E,0x4AEF,{0x9B,0x28,0x61,
0xD8,0x2A,0x1D,0xF7,0x84}};
```

在OnInitDialog()函数中添加初始化COM组件的程序代码：

```
::CoInitialize(NULL);
```

在启动按钮的点选函数中添加启动Google Earth客户端的程序代码：

```
if(m_geApplication.IsInitialized())
{
//客户端未初始化,则创建一个客户端实例
m_geApplication.CreateDispatch(CLSID_GoogleEarth,NULL);//获取客户端框架窗口句柄
OLE_HANDLE m_hMainWnd = m_geApplication.GetMainHwnd();//设置窗口的位置和大小
::SetWindowPos((HWND)m_hMainWnd,HWND_BOTTOM,0,0,800,600,SWP_SHOWWINDOW);
}
else
{
MessageBox(_T("Google Earth初始化失败！"));
}
```

5）编译测试。

### 7.4.2　地图定位功能

下面介绍如何定位某个坐标点的位置。在上面程序的基础上增加地图定位的功能。

1）实现如图7-9所示的界面。

2）给4个文本框赋予相应的变量。

由于要从相应的文本框中提取对应输入的数据，所以需要给这些文本框赋予一些相应类型的变量，以方便获取其中的数据。

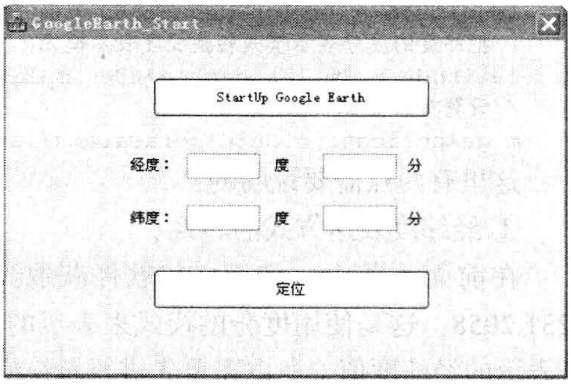


图7-9　定位软件界面

在经度的度的文本框上点选右键选择“Add Variable”选项，在出现的对话框中，先选择“Category”为“Value”，然后选择“Variable type”为“int”，最后，在Variable name框中指定一个变量的名称“m_lon_ degree”，如图7-10

所示。这样就把该文本框和一个int型变量赋值了，在应用中当在该文本框中输入了一个整数，然后使用UpdateData()函数之后，此时“m_lon_degree”变量中的值就是输入的整数，这样就可以方便地获取到在文本框中输入的整数值。

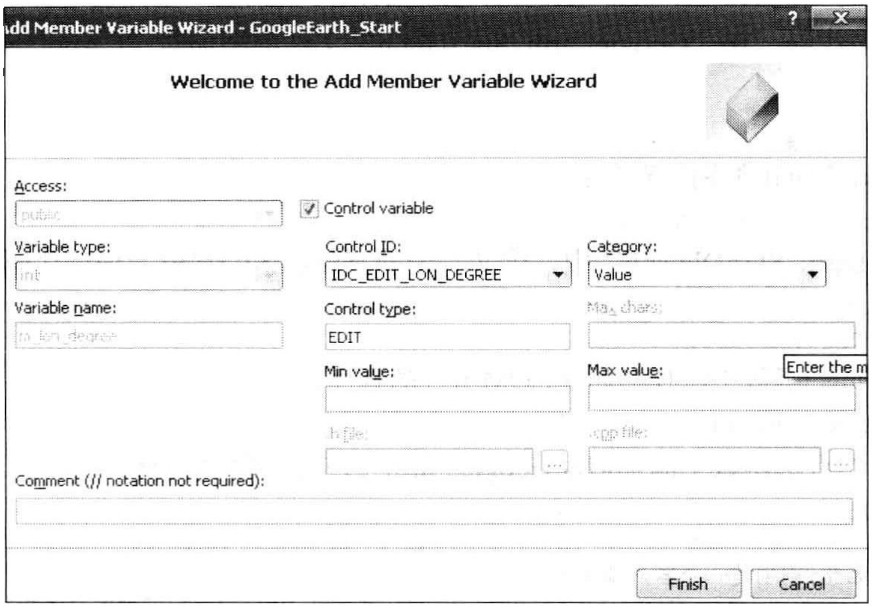



图7-10　给4个文本框赋予相应的变量

使用类似的操作可以为纬度的度输入文本框也添加一个名为“m_lat_degree”的int型变量。

使用类似的操作把经度和纬度的分输入文本框也添加上赋予的变量。只不过需要注意的是，这里在选择变量类型的时候要选择double型。因为这里需要在这两个框中输入小数型的数据。

3）在定位按钮中添加程序代码。

```
UpdateData();      //更新文本框赋予的变数的值
double longitude; //经度
double latitude;  //纬度
//把经度的度分表示模式转换成度表示模式
longitude = (m_lon_degree*60 + m_lon_min)/60.0;
//把纬度的度分表示模式转换成度表示模式
latitude = (m_lat_degree*60 + m_lat_min)/60.0;
//设置视点位置
m_geApplication.SetCameraParams(latitude,longitude,0,1,1000,0,0,0.1);
```

这里有两点需要说明的：

① 经纬度表达方式的转换。

在前面小节中，使用GPS软件获取到的GPS的坐标值。如经度：11355.3403；纬度：2231.2058。这是使用度分的模式来表示的经纬度。而在Google Earth中，只能接收以度的模式来表示的经纬度值，所以需要先进行转换操作才可以。

在说明度分表达模式转换成度表达模式之前，先来说明一下在度分的表达模式下，哪些

位置的数据表示度，哪些位置的数据表示分。以上面的示例为例，113表示度，55.3403表示分。22表示度，31.2058表示分。所以当在软件中输入坐标的时候，也应该这样输入到相应的文本框中。

度分模式转换成度的方法是：度模式的值等于把度分模式中的度乘以60加上分的值，然后除以60。

如果要把度模式的值转换成度分模式，则是一个逆运算。把度模式的值除以60，取到得整数部分就是度分模式中的度的值，余数部分则是分的值。

② SetCameraParams参数含义。

用户通过Google Earth客户端所看到的地图，相当于用户处在一定的高度，用一个照相机所拍摄的图片。在这个过程中，用户所处的高度就是视高，照片的中心就是视底点。该函数的原型如下：

```
    void SetCameraParams(double lat,double lon,double alt,long altMode,double
Range,double Tilt,double Azimuth,double speed)
```

该函数接收8个参数，除了altMode外，其余都是double型。其中，lat表示纬度，lon表示经度，alt表示海拔高度，altMode表示高度值以何种形式描述，range表示视高，titl表示用户视线方向和垂直地平线方向的夹角，其值在0°到90°之间，azimuth即测绘学中的方位角，表示目前视线方向和正北方向的夹角，speed表示移动到该位置的速度，值越大，速度越快。

4）编译程序。执行该软件，在纬度的输入框中输入上面列举的坐标值，效果如图7-11所示。

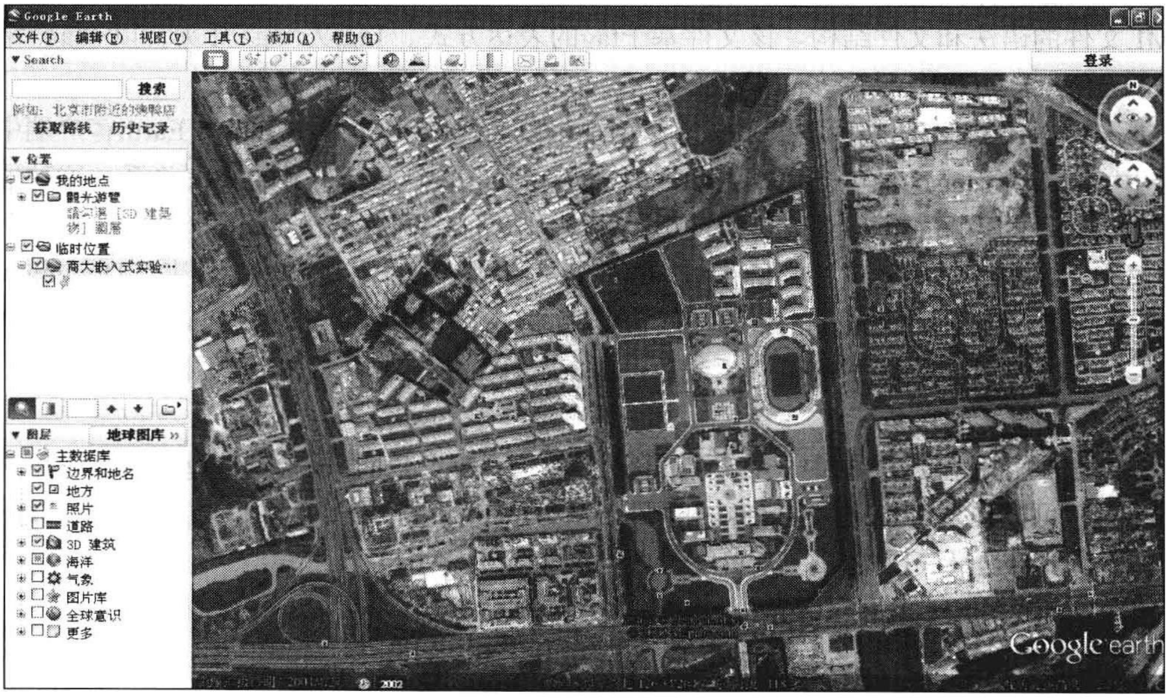



图7-11 Google Earth定位

## 7.4.3 在地图上新增地标

上一小节介绍了如何把视点设置到某一个位置，可以使用这种方法来定位当前的位置，但是在很多的时候希望的是能有一个标志来标出当前的位置，如图7-12所示。

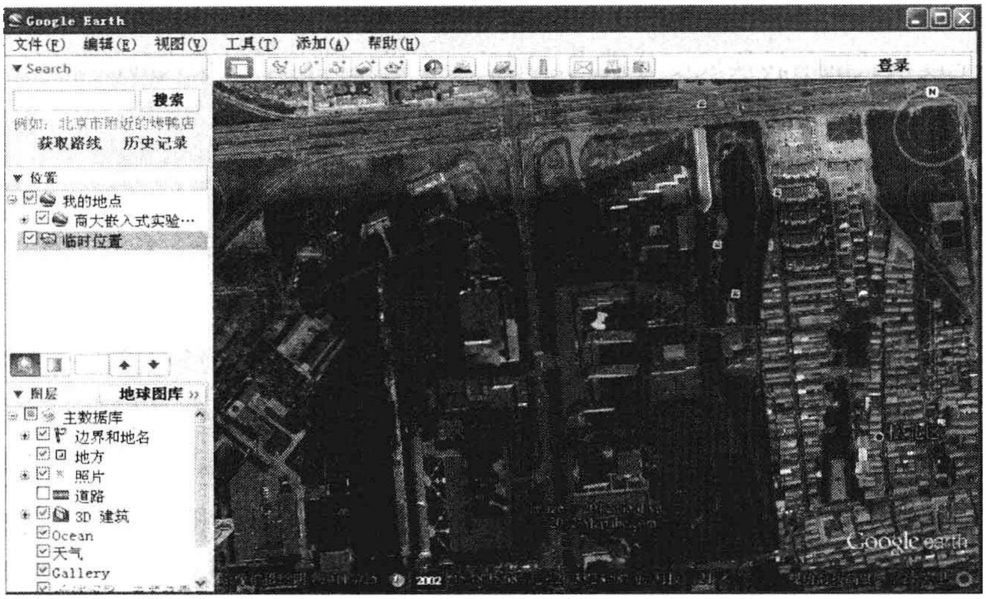



图7-12 用标志来标出当前的位置

要实现这种标记有多种方法，这里介绍的是使用KML文件的方式来实现。KML文件采用了XML文件的语法和文件结构。该文件基于tag的表达方式。Tag以<tag>开头，以</tag>结尾，中间是tag值。KML文件是许多tag构成的集合。

Google Earth定义的tag种类繁多，因而用户可以通过KML文件在Google Earth客户端中显示丰富的内容。鉴于tag这种包含关系，Google Earth的tag集中具有明显的层次关系，这种层次关系是树状的。上层的tag可以包含下层的tag，反之则不能。同一层次的tag可以并列存在于KML文件中。KML的下层主要包括Folder和Document，其中Folder用来创建文件夹，Document用来创建一个目录。Folder和Document的下层包括很多tag，其中最重要的是Placemark。基本上在任何一个KML文件中，用户都可以看到Placemark。Placemark用来表示视图中标定位置的相关信息。

在KML的这种层次结构中，有两点值得注意。第一点是同一层次的tag可以互相包含，如Document中可以包含Folder，Folder中也可以包含Document。第二点是最底层的tag，其首字母为小写。如果不是最底层的tag，其首字母要大写（除了KML）。

使用下面这样一段KML代码可以生成一个地标点。

```
<?xml version="1.0" encoding="UTF-8"?>
<kml >
   <Document>
          <Placemark>
```

```
                <Point>
<coordinates>113.9223383333539,22.52009666677308,0</coordinates>
                </Point>
            </Placemark>
    </Document>
</kml>
```

这是一段很简单的KML代码，可以从代码中看出， <coordinates>113.9223383333539，22.52009666677308，0</coordinates>在控制标记点的坐标，其中113.9223383333539表示经度，22.52009666677308表示纬度。

如果把标记的坐标在地图上标记出来，只要转换两个值即可在上一小节软件的基础上来实现这一功能。这里只需要修改定位按钮中的程序代码即可，程序代码修改如下：

```
UpdateData();      //更新文本框赋予的变数的值
double longitude; //经度
double latitude;  //纬度
//把经度的度分表示模式转换成度表示模式
longitude = (m_lon_degree*60 + m_lon_min)/60.0;
//把纬度的度分表示模式转换成度表示模式
latitude = (m_lat_degree*60 + m_lat_min)/60.0;
//设置视点位置
//m_geApplication.SetCameraParams(latitude,longitude,0,1,1000,0,0,0.1);

CString str_lon;
CString str_lat;
CString str_write;
str_lon.Format(_T("%f"),longitude);
str_lat.Format(_T("%f"),latitude);
str_write = str_lon + _T(",") + str_lat + _T(",0");
char c_write[64];
wcstombs(c_write,str_write,64);
//创建文件
HANDLE hfile = CreateFile(_T("D:\\mark.kml"),GENERIC_WRITE,0,NULL,CREATE_ALWAYS,FILE_
ATTRIBUTE_NORMAL,NULL);
char title[] = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\
<kml >\
   <Document>\
           <Placemark>\
                   <Point>\
                   <coordinates>";
DWORD write;
WriteFile(hfile,title,strlen(title), &write,NULL);
//写坐标
WriteFile(hfile,c_write,strlen(c_write), &write,NULL);
//写结尾
char end[] = "</coordinates>\
             </Point>\
           </Placemark>\
   </Document>\
</kml>";
WriteFile(hfile,end,strlen(end), &write,NULL);
CloseHandle(hfile);
```

```
//汇入KML文件
m_geApplication.OpenKmlFile(_T("D:\\mark.kml"),1);
```

与上一个应用的不同之处在于，增加了一个生成KML代码的步骤和添加该KML文件的步骤。执行该程序，输入前面列举的坐标，效果如图7-13所示。

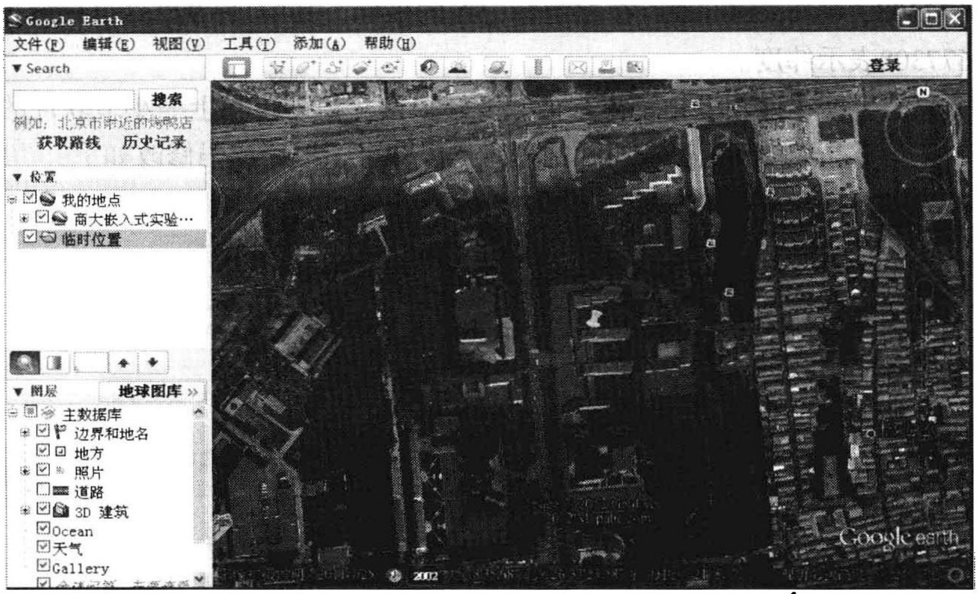



图7-13 标记的坐标在地图上显示出来

# 第8章　鲜奶配送系统

近些年，鲜奶企业开始跨地区、规模化经营，品种不断更新换代。从鲜奶物流配送整体情况来看，随着消费人群的增加、生产规模的扩大、物流配送范围的扩展，鲜奶行业的飞速发展与滞后的配送环节形成了鲜明的反差。由于不同种类的鲜奶保存条件及保质期的不同，对鲜奶的配送提出了更高的要求。针对这种情况，本系统拟设计一套鲜奶配送系统终端。利用Windows CE终端实现鲜奶配送，使配送员能够在终端上完成订单的查询、下载与上传并能够及时调整配送信息。本程序在Visual Studio 2005开发环境中编写与编译，使用C++与MFC作为开发的语言环境。分别在服务器端与Windows CE终端安装SQL Server CE数据库环境。利用SQL Server CE数据库进行数据的存储与交换。

## 8.1　系统总体设计

### 1. 系统总体结构

通过对配送员的需求分析和功能分析，结合该系统的业务，设计出本系统的总体结构图，包含配送模块、传输数据模块、配送调整模块、历史数据查询模块和配送员设置模块。这些功能模块以菜单按钮的形式展示给用户，支持用户通过触屏点击按钮来启动相应的功能，如图8-1所示。

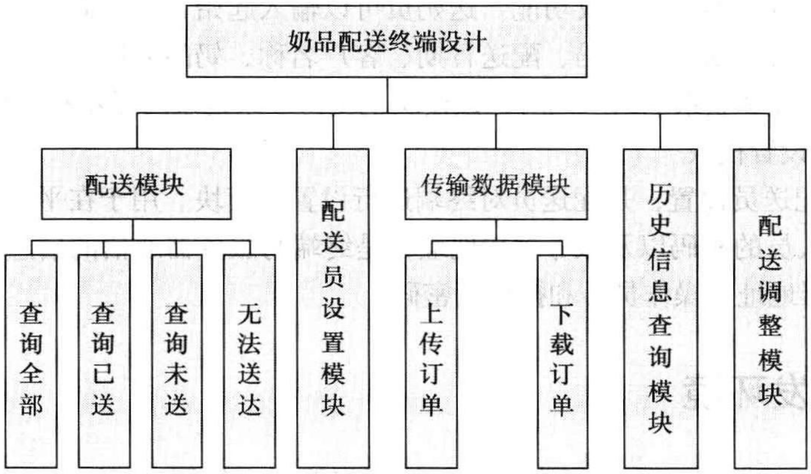



图8-1　Windows CE鲜奶配送终端模块总体框架结构

### 2. 系统功能设计

（1）配送模块

这里设计了配送单查询，查询配送单模块是配送员较多使用的功能模块，用于查询从鲜奶管理信息系统中下发的配送单数据，用户可以输入配送日期、客户名称、客户地址和奶品名称作为条件进行查询，支持模糊查询。并可以点击查询未送、查询已送、无法送达、查询全部的按钮来查看当天配送情况。查询未配送信息后可以进行配送，并修改配送成果，将配送情况更新到本地数据库。本模块不提供配送修改功能，如需对配送成果进行修改，需要到配送调整模块进行。具体查询内容包括客户编码、配送日期、客户名称、奶品名称、奶品数量、家庭地址、客户要求、配送情况和联系电话。同时，该模块支持多条记录的显示和滚动。

（2）传输数据模块

这里设计了订单上传与下载，传输数据模块是配送每日与服务器进行数据更新的功能模块，可从服务器下载当日需配送的数据信息和上传配送结束后的成果信息。并且能够显示下载或上传的数据量为多少条。

（3）配送调整模块

这里设计了送奶员对奶品是否送到的确认操作，是配送员对当日配送信息进行修改的模块。具体查询内容和配送单查询内容一致，同时增加“送到”和“未送到”两个单选按钮，可将已配送或未配送的信息进行修改，如果选择输入“送到”，则需要选择输入配送确认信息，若对该顾客已经完成配送，那么就点击“是”按钮进行确认，反之，则点击“否”按钮；如果选择输入“未送到”，则需要选择输入未配送成功的原因，具体原因包括电子门打不开、奶瓶数量不够、地址信息不正确，其修改行为也将被保存。

（4）历史信息查询模块

这里设计了历史数据查询，是配送员对往日配送信息的查询模块。用于对往日已配送或未配送的信息进行查询，不具备修改功能。送奶员可以输入起始日期和截止日期来作为条件进行查询。具体查询内容包括客户编码、配送日期、客户名称、奶品名称、奶品数量、家庭地址、客户要求、配送情况和联系电话。同时，该模块也支持多条记录的显示和滚动。

（5）配送员设置模块

这里设计了配送员设置，是配送员对终端进行设置的模块。用于在平板电脑上设定本机的使用人员、使用人员的密码以及服务器IP地址，是终端与服务器通信的信息模块。具体的设置内容包括：服务器地址、操作员号和操作员密码。

## 8.2 搭建开发环境

本程序在Visual Studio 2005开发环境中编写与编译，使用C++与MFC作为开发的语言环境。分别在服务器端与Windows CE终端安装SQL Server CE数据库环境。利用SQL Server CE数据库进行数据的存储与交换。

### 1. 服务器端SQL Server 2005数据库的安装

在服务器端安装SQL Server 2005，设置用户名为sa，密码为123456。

### 2. SQL CE数据库的安装与配置

SQL CE包括两个层面的安装，一是客户端安装，即在嵌入式或移动设备中安装SQL CE 3.0环境；二是服务器端安装，即在服务器端实现服务器同客户端设备的SQL CE数据同步、订阅等工作。如果只是在客户端做应用，而不需要和服务器进行数据同步等工作，那么可以不用安装配置服务器环境。

（1）SQL CE 3.0客户端安装过程

SQL CE 3.0安装文件可以从微软网站进行下载，也可以从Visual Studio 2005安装之后的目录中获取。在安装VS 2005时，会包含SQL CE 3.0安装文件，默认目录为C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\wce500\armv4i。该目录下包含了9个CAB安装文件。这9个安装文件对应3组：一组是针对smartphone智能手机的；一组是针对pocket pc智能PDA的；一组是针对Windows CE通用系统的。

将sqlce30.ppc.wce5.armv4i.CAB、sqlce30.repl.ppc.wce5.armv4i.CAB和sqlce30.dev.CHS.ppc.wce5.armv4i.CAB这3个文件复制到PDA中，并依次进行安装。

其中sqlce30.ppc.wce5.armv4i.CAB安装文件是SQL CE 3.0核心安装文件；sqlce30.repl.ppc.wce5.armv4i.CAB安装文件是服务于复制、订阅等相关服务；sqlce30.dev.CHS.ppc.wce5.armv4i.CAB是SQL CE查询分析器的安装程序，以便于用户对SQL CE进行管理和操作，该查询分析器的默认目录是\\Program Files\SQL Mobile\CN\isqlw30.exe。

如果在Windows CE 6.0系统中安装这三个文件时，可能会提示这3个文件安装有可能不兼容，是否继续安装。我们选择继续安装，因为这种情况不影响SQL CE 3.0的使用。安装完毕后，用户可以通过打开SQL CE查询分析器来建立SQL CE数据库以及相关数据表，并实现数据存储和查询等相关功能。

（2）SQL CE 3.0服务器端的安装与配置

服务器可以是普通PC，也可以是服务器。操作系统可以是Windows XP SP2、Windows 2000 Server、Windows 2003 Server等。

若SQL CE 3.0应用程序要求在Microsoft SQL Server数据库和SQL CE数据库之间交换数据，则必须在服务器上安装如下软件：

Microsoft Internet information Server（IIS）。

SQL Server 2005或SQL Server 2000。

SQL CE 3.0服务器工具。该工具的默认目录位于C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0下的Sqlce30setupcn.msi。安装后，系统中将会有“配置WEB同步向导”工具。

### 3. SQL CE 3.0连接配置

本节就以示例的形式介绍如何在客户端和服务器端建立代理连接，以便客户端能够和服务

器端进行数据交互。

1）选择“开始”→“Microsoft SQL Server 2005 Mobile Edition”→“配置Web同步向导”选项，打开SQL CE连接配置管理工具，如图8-2所示。

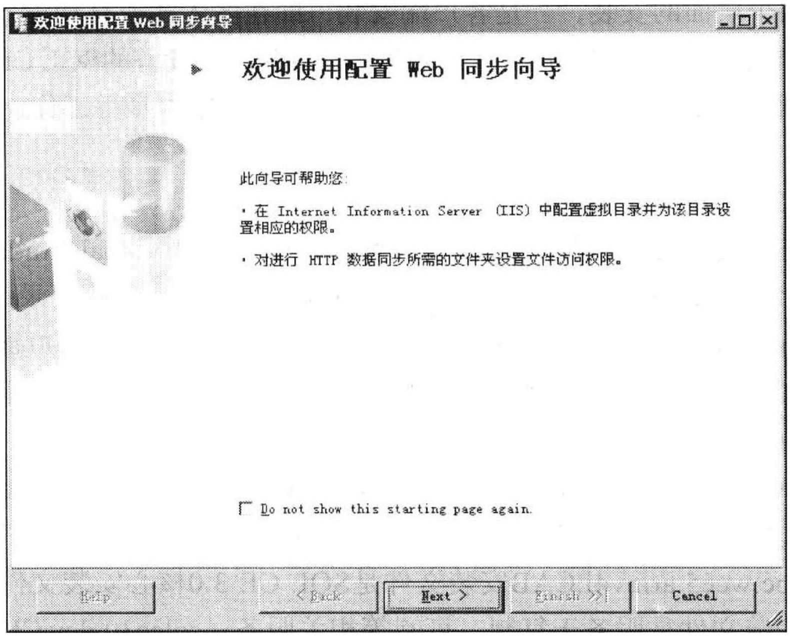



图8-2　SQL CE连接配置管理工具

2）单击“Next”按钮，进入如图8-3所示的“订阅服务器类型”界面，这里选择SQL Server Mobile Edition。

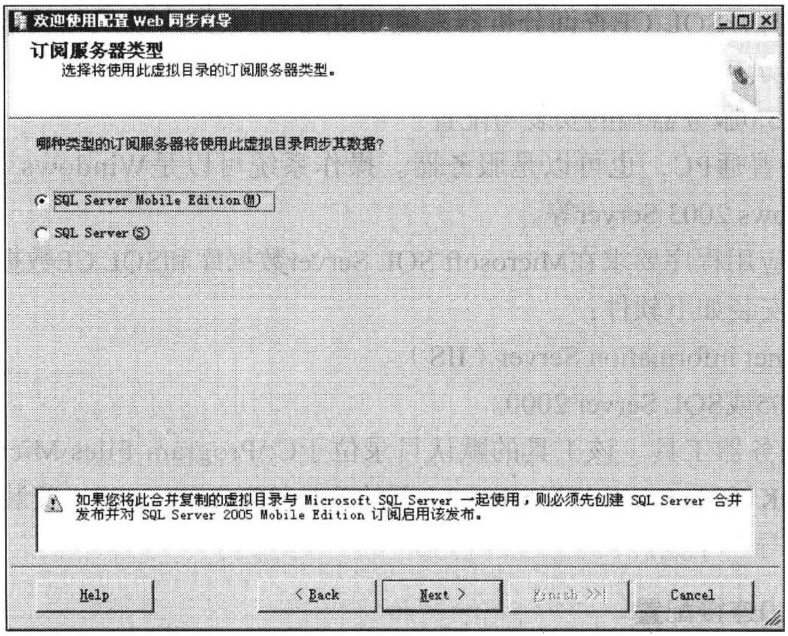



图8-3　订阅服务器类型

3）单击“Next”按钮，进入设置“Web服务器”界面，这里输入运行IIS的计算机名，输入本计算机的网络名（默认即可），选择“创建新的虚拟目录”单选按钮，选中“默认网站”，具体设置如图8-4所示。

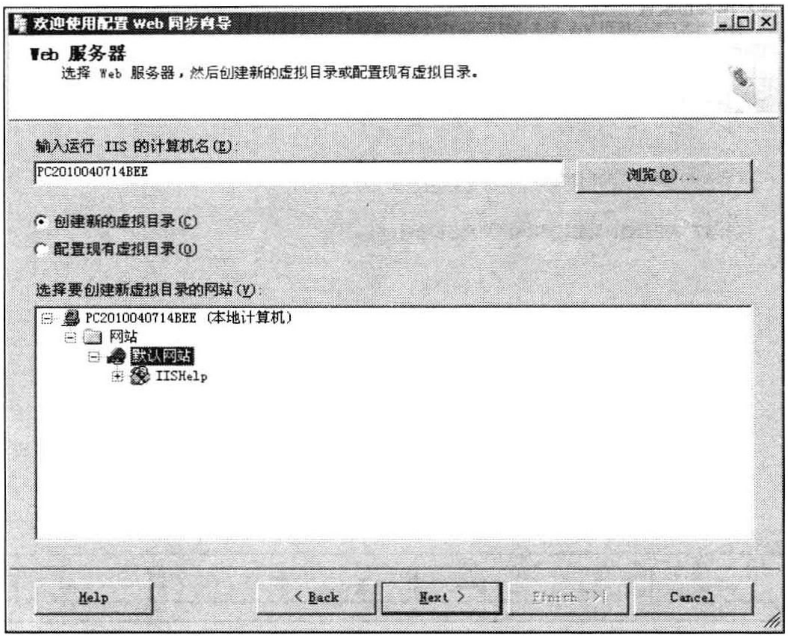



图8-4　设置Web服务器

4）单击“Next”按钮，进入如图8-5所示的设置“虚拟目录信息”界面，这里将别名设置为sqlce，路径设置为默认。这个默认路径包含了sqlcesa30.dll文件。

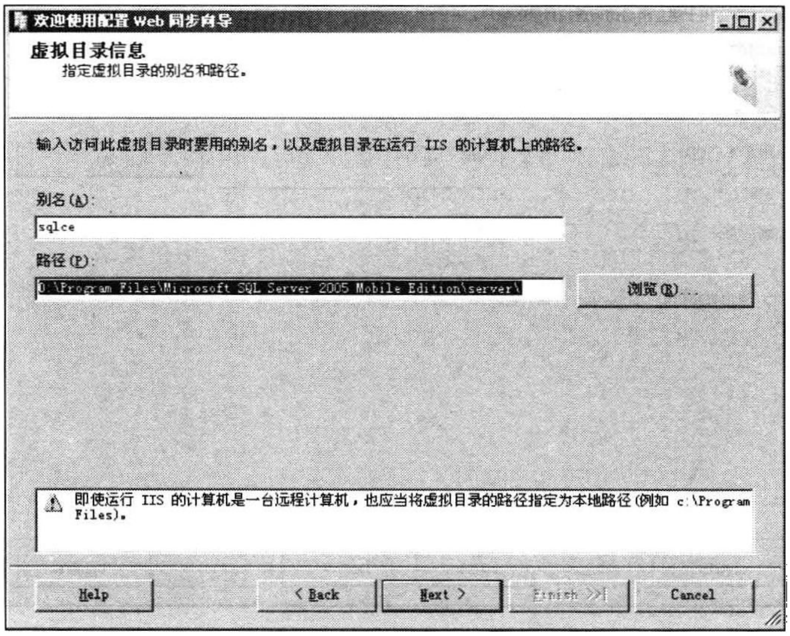



图8-5　设置虚拟目录信息

5）单击“Next”按钮，进入设置“安全通信”界面，选择默认项。单击“Next”按钮，进入设置“客户端身份验证”界面，设置为匿名方式，如图8-6所示。

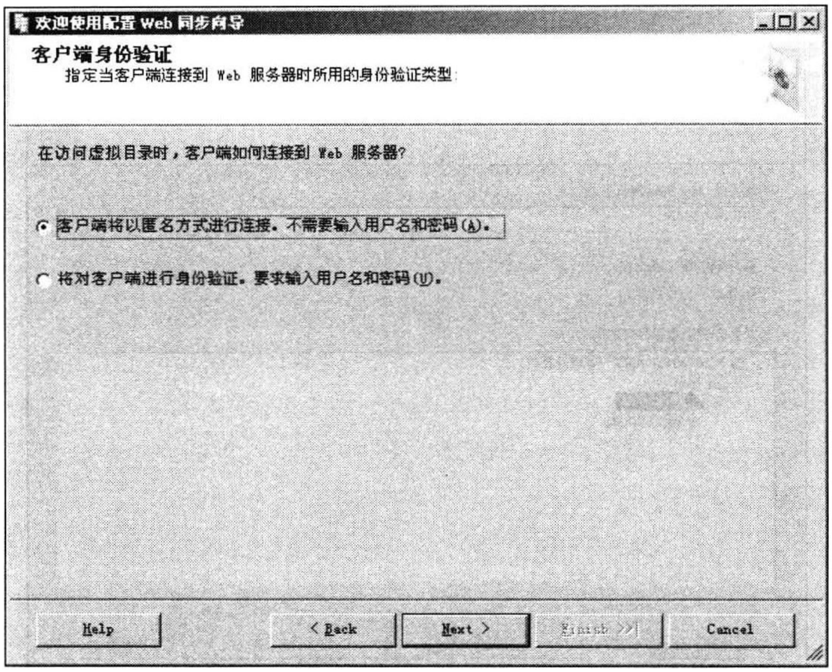



图8-6 设置客户端身份验证

6）单击“Next”按钮，设置“匿名访问”参数，默认即可，如图8-7所示。

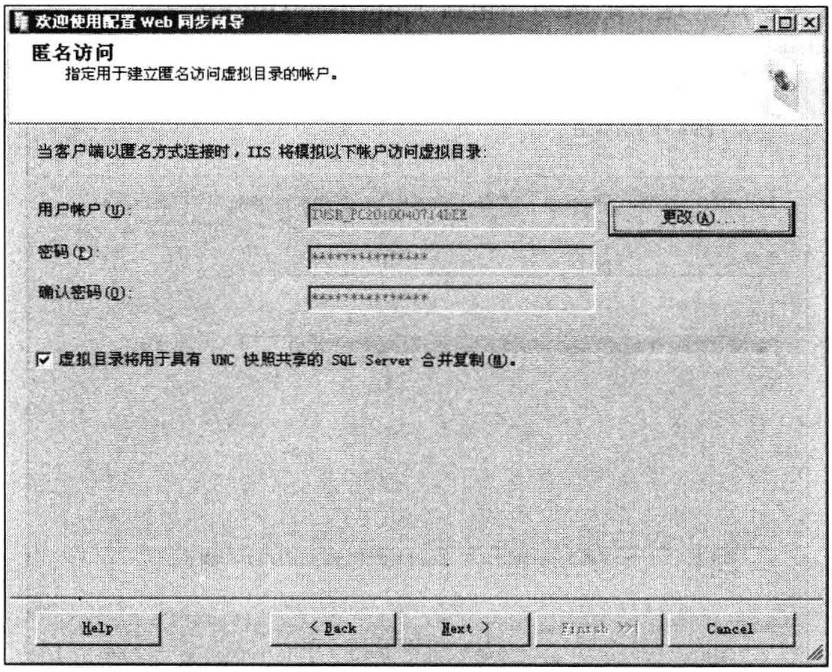



图8-7 设置匿名访问参数

7）单击“Next”按钮，进入设置“快照共享访问”界面，这里需要一个共享文件目录。本示例在C盘建立了一个新目录snapshot，并将其设置成共享，具备读写权限，共享名为snapshot，因此在共享目录中输入\\计算机名\snapshot，如图8-8所示。

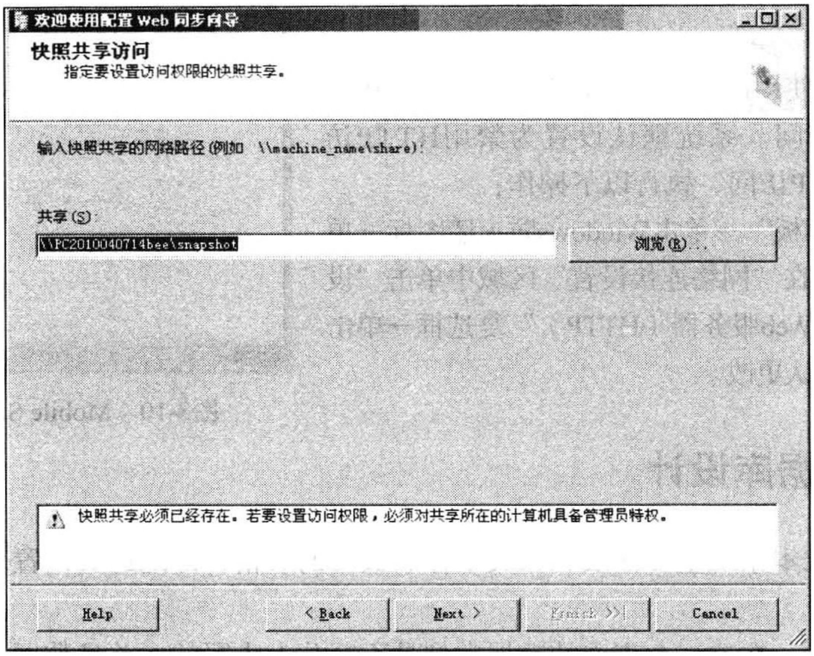



图8-8　设置快照共享访问

8）单击“Next”按钮便可以完成配置，配置状态如图8-9所示。

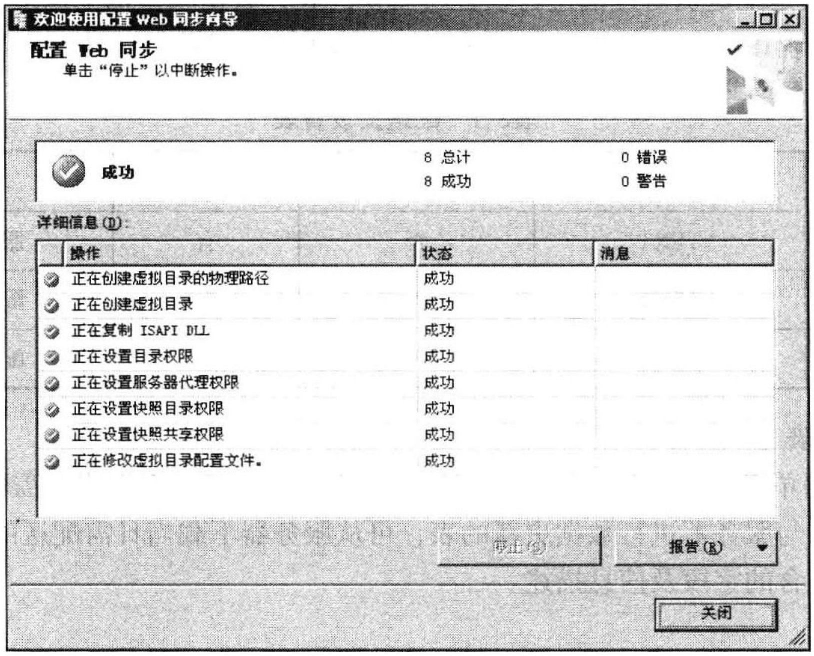



图8-9　配置状态

在SQL CE 3.0连接配置结束后，有一个简单的方法可以判断配置是否成功，即在浏览器中输入SQL Server CE Server Agent（sqlcesa30.dll）的网络路径，当出现如图8-10所示的界面时，表示配置成功。在该例中LJD-5为机器名，sqlce30为虚拟目录别名。

9）如果用户使用的是Windows XP SP2运行IIS，必须启用HTTP访问。系统默认设置为禁用HTTP访问。若要启动HTTP访问，执行以下操作：

打开“控制面板”→单击Windows防火墙图标→单击“高级”按钮→在“网络连接设置”区域中单击“设置”按钮→选中“Web服务器（HTTP）”复选框→单击“确定”按钮以确认更改。

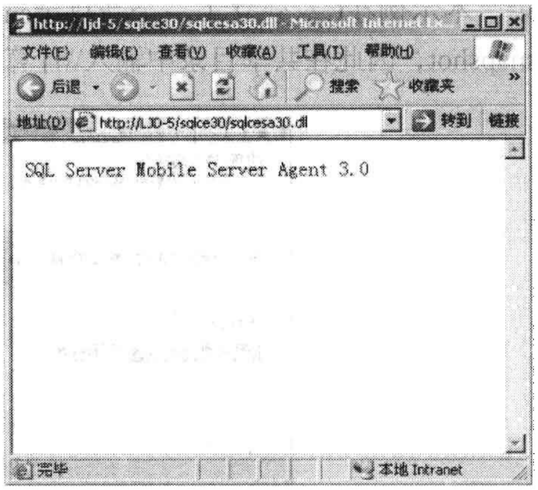


图8-10 Mobile Server验证成功

## 8.3 系统数据库设计

根据系统需求本系统主要涉及三张关于存放数据的表格，即配送员设置表、配送日报表和配送日报备份表。

这里说明的是，在这三个表中的数据类型是不区分大小写的。并且数据库设计的表只是一个参考而不是完全相同的，数据库表的设计还需要在实际中具体情况具体分析。

### 1. 配送员设置表（setInfo）

配送员设置表是配送员对终端进行设置所需要的表，数据项包括配送员编号和密码，这一表中也提供了服务器IP项。表8-1所示为该表中所包含的字段及信息描述。

表8-1 配送员设置表

| 字 段 名 | 类 型 | 是否为空 | 主 键 | 备 注 |
|---|---|---|---|---|
| address | TEXT | 否 | 有 | 服务器IP地址 |
| username | TEXT | 否 | 有 | 配送员号 |
| password | TEXT | 否 | 有 | 配送员密码 |

### 2. 配送日报表（t_psrb）

由于查询配送单模块是配送员使用较多的功能模块，这里可查询当日需配送的数据信息。配送日报表是每日与服务器进行数据更新的表，可从服务器下载当日需配送的数据信息。表8-2所示为该表中所包含的字段及信息描述。

表8-2 配送日报表

| 字 段 名 | 类 型 | 是否为空 | 主键 | 备 注 |
|---|---|---|---|---|
| id | INTEGER | 否 | 有 | 数据ID |
| psrq | TEXT | 否 | 有 | 配送日期 |
| khid | TEXT | 否 | 有 | 客户ID |
| dj_p | TEXT | 是 | 无 | 单价 |
| pssl | INTEGER | 是 | 无 | 配送数量 |
| khmc | TEXT | 是 | 无 | 客户名称 |
| npmc | TEXT | 是 | 无 | 奶品名称 |
| khdz | TEXT | 是 | 无 | 客户地址 |
| psyq | TEXT | 是 | 无 | 配送要求 |
| psyh | TEXT | 是 | 无 | 配送员号 |
| bz_isdown | INTEGER | 是 | 无 | 已下载标志 |
| bz_isup | INTEGER | 是 | 无 | 已上传标志 |
| psy_psrq | TEXT | 是 | 无 | 配送日期 |
| psy_lasttime | TEXT | 是 | 无 | 最后修改日期 |
| psy_xgcs | INTEGER | 是 | 无 | 修改次数 |
| psqk | TEXT | 是 | 无 | 配送情况 |
| psbz | INTEGER | 是 | 无 | 配送标志 |

### 3. 配送日报备份表（t_psrb_save）

由于配送员对往日的配送信息需要查询，因此我们设计了配送日报备份表，可对往日已配送或未配送的信息进行查询，它与配送日报表的结构和内容是一样的（可参考8-2），不同的是配送日报表保存的是当日从服务器下载的需要配送的数据信息，而配送日报表保存的是往日的配送信息情况。假如，我们已经完成了今天的配送任务，检查无误后，上传了配送结束后的信息，那么配送日报表当前就没有数据了，我们需要到配送日报备份表中去查询我们已经上传的当日和往日的配送信息。

## 8.4 系统初始化

在系统的初始化过程中，首先建立数据库环境，具体代码如下：

```
#include <ssceoledb30.h>
CoInitializeEx(NULL,COINIT_MULTITHREADED)
   HRESULT hr = NOERROR;  // Error code reporting
   HANDLE hFind;                              // File handle
   WIN32_FIND_DATA      FindFileData;         // The file structure description
```

```
        // If database exists,open it,
        // Otherwise,create a new database,insert sample data.
        //
        hFind = FindFirstFile(dbname, &FindFileData);
        if(INVALID_HANDLE_VALUE == hFind)
        {
                DeleteFile(dbname);

    ////登录系统使用用户ID,用户名与用户密码
                LPTSTR CreateSql_Users          = L"CREATE TABLE Users(UserName NVARCHAR(20),
UserKey NVARCHAR(20),AddIP NVARCHAR(20))";

///////////////////////配送日报表/////////////////////////////
////配送标志psbz = 0未送1已送2无法送达
LPTSTR CreateSql_PSRB              = L"CREATE TABLE PSRB(ID INT,psrq NVARCHAR(20),kuid NVARCHAR(20),
dj_p NVARCHAR(20),pssl INT,kumc NVARCHAR(20),npmc NVARCHAR(20),khdz NVARCHAR(20),khdh
NVARCHAR(20),psyq NVARCHAR(20),psyh NVARCHAR(20),bz_isdown INT,bz_isup INT,psy_psrq
NVARCHAR(20),psy_lasttime NVARCHAR(20),psy_xgcs INT,psqk NVARCHAR(20),psbz INT)";

                ///////////////////////配送日报备份表/////////////////////////////
                LPTSTR CreateSql_PSRB_SAVE = L"CREATE TABLE PSRB_SAVE(ID INT,psrq NVARCHAR(20),
kuid NVARCHAR(20),dj_p NVARCHAR(20),pssl INT,kumc NVARCHAR(20),npmc NVARCHAR(20),khdz
NVARCHAR(20),psyq NVARCHAR(20),psyh NVARCHAR(20),bz_isdown INT,bz_isup INT,psy_psrq
NVARCHAR(20),psy_lasttime NVARCHAR(20),psy_xgcs INT,psqk NVARCHAR(20),psbz INT)";

                // Create  database
                hr = CreateDatabase(dbname);

                hr = ConnectDB(dbname);

                hr = GetRowset(CreateSql_Users);

                DisconnectDB();

                ConnectDB(dbname);
                hr = GetRowset(CreateSql_PSRB);
                DisconnectDB();

                ConnectDB(dbname);
                hr = GetRowset(CreateSql_PSRB_SAVE);
                DisconnectDB();
        }
```

# 8.5 终端应用程序设计

## 8.5.1 登录界面

首先根据用户名与密码登录系统，如图8-11所示。在该设计中，提供了简单的注册功能，单击“注册”按钮后进入注册界面，通过输入用户名、密码与服务器IP地址来完成用户的注

册，如图8-12所示。

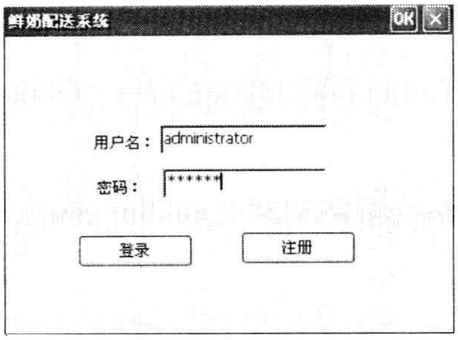



图8-11 登录界面

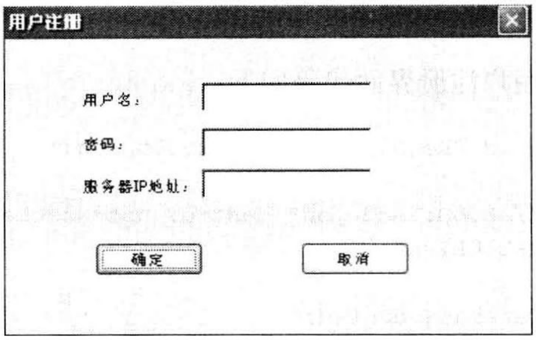



图8-12 注册用户

登录界面的代码如下：

```
HRESULT hr = NOERROR;
Database db_log;

hr = db_log.ConnectDB(L"MilkDeliver.sdf");

CString tmp1,tmp2;
GetDlgItem(LOG_EDIT_USERNAME)->GetWindowTextW(tmp1);
const wchar_t* Username = tmp1;

GetDlgItem(LOG_EDIT_PASSWORD)->GetWindowTextW(tmp2);
const wchar_t* Userkey = tmp2;

TCHAR szBuffer[1024];
wsprintf(szBuffer,TEXT("SELECT UserKey FROM Users WHERE UserName = '%s'"),Username);

hr = db_log.GetRowset(szBuffer);

const wchar_t* TTSInfo = db_log.g_TTSInfo;

if(0 == _wcsicmp((WCHAR*)L"",(WCHAR*)(TTSInfo)))
{
   MessageBox(L"用户名不存在！");
   goto Exit;
}
else if(0 == _wcsicmp((WCHAR*)Userkey,(WCHAR*)(TTSInfo)))
{

   CMainDlg mainf = new CMainDlg;
   mainf.DoModal();
}
else
{
   MessageBox(L"密码错误！请重新输入！");
   goto Exit;
```

```
}
Exit:
   db_log.DisconnectDB();
```

用户注册界面代码如下:

```
void CRegDlg::OnBnClickedBtnReg()
{
// TODO: Add your control notification handler code here
HRESULT hr;

Database db_reg;
db_reg.ConnectDB(L"MilkDeliver.sdf");
CString str_ipadd,str_name,str_key;
GetDlgItem(REG_EDIT_USERNAME)->GetWindowTextW(str_name);
GetDlgItem(REG_EDIT_PASSWORD)->GetWindowTextW(str_key);
GetDlgItem(REG_EDIT_IPADD)->GetWindowTextW(str_ipadd);
TCHAR sqlstr[1024],sqlstr2[1024];
wsprintf(sqlstr,TEXT("select * from Users where UserName='%s'"),str_name);
          hr = db_reg.GetRowset(sqlstr);
                 if(0 == _wcsicmp(L"",db_reg.g_TTSInfo))
          {
                                db_reg.DisconnectDB();
   wsprintf(sqlstr2,TEXT("insert into Users values('%s','%s','%s')"),str_name,str_key,str_ipadd);

   db_reg.ConnectDB(L"MilkDeliver.sdf");
   hr = db_reg.GetRowset(sqlstr2);

   if(SUCCEEDED(hr))
   {
          MessageBox(L"注册成功!");
   }
          }
          else
    {
                 MessageBox(L"用户名已存在!");
    }
Exit:
   db_reg.DisconnectDB();
}
```

## 8.5.2 主界面

主界面是程序的入口界面，如图8-13所示，包含了配送查询模块、数据传输模块、配送调整模块、历史查询模块和配送员设置模块，这些功能模块以菜单按钮的形式展示给用户，支持用户通过触屏点击按钮来启动相应的功能。

图8-13 鲜奶配送系统主界面

## 8.5.3　配送模块

配送模块中设计了配送单查询，用于查询从鲜奶管理信息系统中下发的配送单数据，用户可以输入相应的条件进行查询，支持模糊查询。具体查询内容包括客户编码、订单号、客户名称、奶品名称、数量、家庭地址、客户要求和联系电话等。支持多条记录的显示和滚动描述。配送单查询的页面效果如图8-14所示。

查询配送单模块是配送员使用较多的功能模块，可查询当日需配送的数据信息，包括查询未送、查询已送、查询全部及查询无法送达。需要注意的是本模块不提供配送修改功能，如需对配送成果进行修改，需要到配送调整模块进行。下面将对这四个查询功能做一下简要说明。

### 1. 查询未送

如图8-15显示了详细的未配送数据，用户可以通过点击方式的查询到指定的信息。

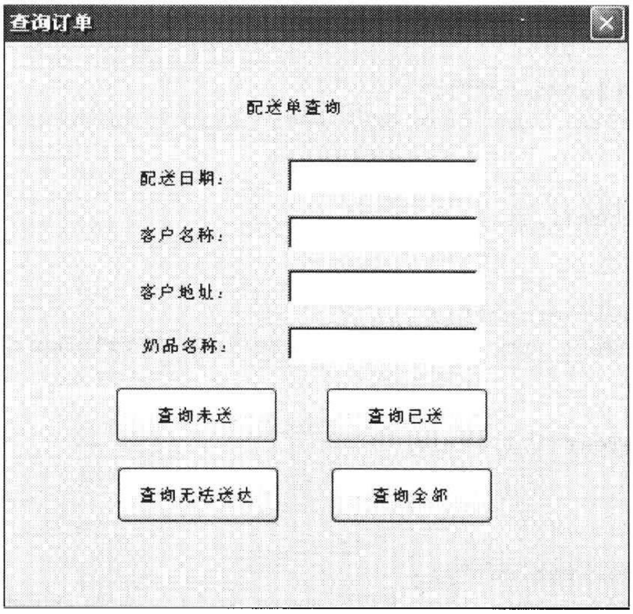



图8-14　配送单查询

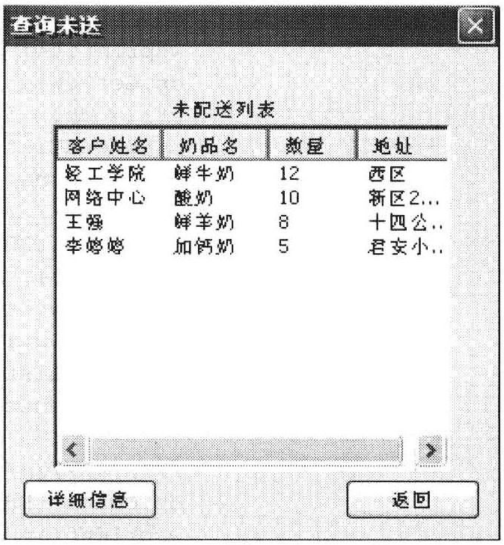



图8-15　未配送列表

该功能模块的代码如下：

```
BOOL CQueryUnsendDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   m_UnsendList.InsertColumn(0,L"客户姓名",LVCFMT_IMAGE|LVCFMT_LEFT,60);
   m_UnsendList.InsertColumn(1,L"奶品名",LVCFMT_IMAGE|LVCFMT_LEFT,60);
   m_UnsendList.InsertColumn(2,L"数量",LVCFMT_IMAGE|LVCFMT_LEFT,50);
   m_UnsendList.InsertColumn(3,L"地址",LVCFMT_IMAGE|LVCFMT_LEFT,60);

   Database db;
   db.ConnectDB(L"MilkDeliver.sdf");
```

```
CString str;
TCHAR sql[1024];

//客户ID
wsprintf(sql,TEXT("select count(1)from PSRB"));
HRESULT hr = db.GetRowset(sql);

int rownumber = *db.g_TTSInfo - 48;//db.gTTSInfo是char值即ASCII码,转换为int型需要-48。即总条数
db.DisconnectDB();
int i=0;
int kuid = 1;
do
{
        db.ConnectDB(L"MilkDeliver.sdf");
        wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d and psbz = 0"),kuid);
        HRESULT hr = db.GetRowset(sql);
        if(0 != _wcsicmp(L"",db.g_TTSInfo))
        {
                db.DisconnectDB();
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_UnsendList.InsertItem(i,str);
                db.DisconnectDB();
                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_UnsendList.SetItemText(i,1,str);
                db.DisconnectDB();

                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_UnsendList.SetItemText(i,2,str);
                db.DisconnectDB();

                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_UnsendList.SetItemText(i,3,str);
                db.DisconnectDB();

                i++;
        }
        else
```

```
                db.DisconnectDB();
            kuid++;
        }
        while(kuid<=rownumber);
        return TRUE;  // 除非将焦点设置到控件,否则返回TRUE
    }
```

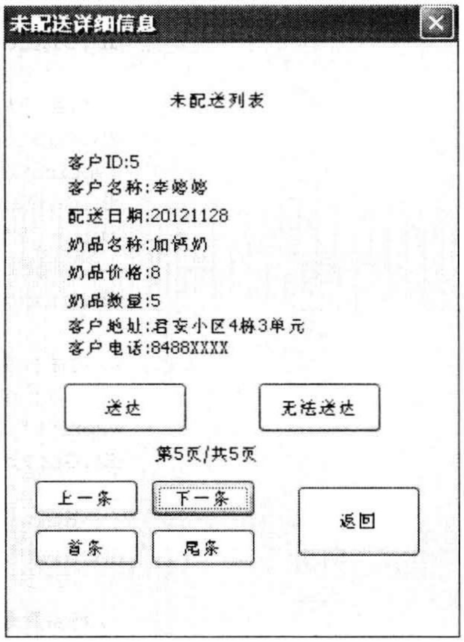



图8-16 配送单详细信息未配送

在图8-15中，如果想要查询更为详细的内容，用户可以直接点击指定的记录，程序将会显示详细配送数据的内容，具体界面如图8-16所示。

该功能模块代码如下：

```
BOOL CQueryUnsendDeDlg::OnInitDialog()
{
    CDialog OnInitDialog();

    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");
    CString str;
    TCHAR sql[1024];

    //客户ID
    wsprintf(sql,TEXT("select  count(1)from
PSRB"));
    HRESULT hr = db.GetRowset(sql);

    rownumber = *db.g_TTSInfo - 48;//db.gTTSInfo是char值即ASCII码,转换为int型需要-48。即总条数
    db.DisconnectDB();

    for(int i=1;i<rownumber+1;i++)
    {
            //客户ID
            db.ConnectDB(L"MilkDeliver.sdf");
            //CString ii = i;
            wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d and psbz = 0"),i);
            HRESULT hr = db.GetRowset(sql);
            if(0 != _wcsicmp(L"",db.g_TTSInfo))
            {
                str = L"客户ID:"+CString(db.g_TTSInfo);
                ((CStatic*)GetDlgItem(IDC_STATIC_ID))->SetWindowTextW(str);
                db.DisconnectDB();

                //客户名称
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d and psbz = 0"),i);
                db.GetRowset(sql);
                str = L"客户名称:"+CString(db.g_TTSInfo);
                ((CStatic*)GetDlgItem(IDC_STATIC_CUSNAME))->SetWindowTextW(str);
                db.DisconnectDB();

                //配送日期
                db.ConnectDB(L"MilkDeliver.sdf");
```

```
wsprintf(sql,TEXT("select psrq from PSRB where kuid = %d and psbz = 0"),i);
db.GetRowset(sql);
str = L"配送日期:"+CString(db.g_TTSInfo);
((CStatic*)GetDlgItem(IDC_STATIC_DATE))->SetWindowTextW(str);
db.DisconnectDB();

//奶品名称
db.ConnectDB(L"MilkDeliver.sdf");
wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d and psbz = 0"),i);
db.GetRowset(sql);
str = L"奶品名称:"+CString(db.g_TTSInfo);
((CStatic*)GetDlgItem(IDC_STATIC_MILKNAME))->SetWindowTextW(str);
db.DisconnectDB();

//奶品价格
db.ConnectDB(L"MilkDeliver.sdf");
wsprintf(sql,TEXT("select dj_p from PSRB where kuid = %d and psbz = 0"),i);
db.GetRowset(sql);
str = L"奶品价格:"+CString(db.g_TTSInfo);
((CStatic*)GetDlgItem(IDC_STATIC_PRICE))->SetWindowTextW(str);
db.DisconnectDB();

//奶品数量
db.ConnectDB(L"MilkDeliver.sdf");
wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d and psbz = 0"),i);
db.GetRowset(sql);
str = L"奶品数量:"+CString(db.g_TTSInfo);
((CStatic*)GetDlgItem(IDC_STATIC_NUMBER))->SetWindowTextW(str);
db.DisconnectDB();

//客户地址
db.ConnectDB(L"MilkDeliver.sdf");
wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d and psbz = 0"),i);
db.GetRowset(sql);
str = L"客户地址:"+CString(db.g_TTSInfo);
((CStatic*)GetDlgItem(IDC_STATIC_ADDRESS))->SetWindowTextW(str);
db.DisconnectDB();

//客户电话
db.ConnectDB(L"MilkDeliver.sdf");
wsprintf(sql,TEXT("select khdh from PSRB where kuid = %d and psbz = 0"),i);
db.GetRowset(sql);
str = L"客户电话:"+CString(db.g_TTSInfo);
((CStatic*)GetDlgItem(IDC_STATIC_TEL))->SetWindowTextW(str);
db.DisconnectDB();

CusID = i;

wsprintf(sql,TEXT("第%d页/共%d页"),CusID,rownumber);

str = sql;
//str = L"第"+_T(""+CusID)+L"条/共"+_T(rownumber+"")+L"条";
```

```
                ((CStatic*)GetDlgItem(IDC_STATIC_INDEX))->SetWindowTextW(str);

                goto Exit;
            }
            else
            {
                if(i==rownumber)
                {
                    MessageBox(L"无未配送资料");
                    db.DisconnectDB();
                    goto Exit;
                }
            }
    }


    Exit:

    return TRUE;
}
//"送达"按钮代码
void CQueryUnsendDeDlg::OnBnClickedBtnRecieve()
{
    // TODO: Add your control notification handler code here
    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");
    TCHAR sql[1024];

    wsprintf(sql,TEXT("update PSRB set psbz = 1 where kuid = %d"),CusID);
    HRESULT hr = db.GetRowset(sql);
    if(hr==NOERROR)
    {
        MessageBox(L"奶品成功送达！");
    }
    db.DisconnectDB();

}

//"无法送达"代码
void CQueryUnsendDeDlg::OnBnClickedBtnUnrecieve()
{
    // TODO: Add your control notification handler code here
    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");
    TCHAR sql[1024];

    wsprintf(sql,TEXT("update PSRB set psbz = 2 where kuid = %d"),CusID);
    HRESULT hr = db.GetRowset(sql);
    if(hr==NOERROR)
    {
        MessageBox(L"奶品添加至无法送达列表！");
    }
```

```
    db.DisconnectDB();
}
// "下一条" 按钮代码
void CQueryUnsendDeDlg::OnBnClickedBtnNext()
{
    // TODO: Add your control notification handler code here
    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");
    CString str;
    TCHAR sql[1024];

    for(int i=CusID+1;i<rownumber+1;i++)
    {
            //客户ID
            db.ConnectDB(L"MilkDeliver.sdf");
            //      CString ii = i;
            wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d and psbz = 0"),i);
            HRESULT hr = db.GetRowset(sql);
            if(0 != _wcsicmp(L"",db.g_TTSInfo))
            {
                    str = L"客户ID:"+CString(db.g_TTSInfo);
                    ((CStatic*)GetDlgItem(IDC_STATIC_ID))->SetWindowTextW(str);
                    db.DisconnectDB();

                    //客户名称
                    db.ConnectDB(L"MilkDeliver.sdf");
                    wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d and psbz = 0"),i);
                    db.GetRowset(sql);
                    str = L"客户名称:"+CString(db.g_TTSInfo);
                    ((CStatic*)GetDlgItem(IDC_STATIC_CUSNAME))->SetWindowTextW(str);
                    db.DisconnectDB();

                    //配送日期
                    db.ConnectDB(L"MilkDeliver.sdf");
                    wsprintf(sql,TEXT("select psrq from PSRB where kuid = %d and psbz = 0"),i);
                    db.GetRowset(sql);
                    str = L"配送日期:"+CString(db.g_TTSInfo);
                    ((CStatic*)GetDlgItem(IDC_STATIC_DATE))->SetWindowTextW(str);
                    db.DisconnectDB();

                    //奶品名称
                    db.ConnectDB(L"MilkDeliver.sdf");
                    wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d and psbz = 0"),i);
                    db.GetRowset(sql);
                    str = L"奶品名称:"+CString(db.g_TTSInfo);

                    ((CStatic*)GetDlgItem(IDC_STATIC_MILKNAME))->SetWindowTextW(str);
                    db.DisconnectDB();

                    //奶品价格
                    db.ConnectDB(L"MilkDeliver.sdf");
                    wsprintf(sql,TEXT("select dj_p from PSRB where kuid = %d and psbz = 0"),i);
                    db.GetRowset(sql);
```

```
            str = L"奶品价格:"+CString(db.g_TTSInfo);
            ((CStatic*)GetDlgItem(IDC_STATIC_PRICE))->SetWindowTextW(str);
            db.DisconnectDB();

            //奶品数量
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d and psbz = 0"),i);
            db.GetRowset(sql);
            str = L"奶品数量:"+CString(db.g_TTSInfo);
            ((CStatic*)GetDlgItem(IDC_STATIC_NUMBER))->SetWindowTextW(str);
            db.DisconnectDB();

            //客户地址
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d and psbz = 0"),i);
            db.GetRowset(sql);
            str = L"客户地址:"+CString(db.g_TTSInfo);
            ((CStatic*)GetDlgItem(IDC_STATIC_ADDRESS))->SetWindowTextW(str);
            db.DisconnectDB();

            //客户电话
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select khdh from PSRB where kuid = %d and psbz = 0"),i);
            db.GetRowset(sql);
            str = L"客户电话:"+CString(db.g_TTSInfo);
            ((CStatic*)GetDlgItem(IDC_STATIC_TEL))->SetWindowTextW(str);
            db.DisconnectDB();

            CusID = i;

            wsprintf(sql,TEXT("第%d页/共%d页"),CusID,rownumber);
            str = sql;
            ((CStatic*)GetDlgItem(IDC_STATIC_INDEX))->SetWindowTextW(str);

            goto Exit;
        }
        else
        {
            if(i==rownumber)
            {
                MessageBox(L"已到最后一条未配送记录");
                db.DisconnectDB();
                goto Exit;
            }
        }
    }
Exit:
;
}
```

其他按钮代码类似，就不一一赘述。

通过这个界面，用户可以点击“送达”或者“未送达”按钮以改变该配送的状态。例如，

点击“送达”按钮将会出现如图8-17所示的送达确认界面。在配送确认界面里如果用户点击“是”按钮，程序将会记录该操作，将本地数据库中的记录状态修改为“已送达”；假如点击“未送达”按钮将会出现如图8-18所示的未送达确认界面，并且要填写未配送成功原因，包括电子门打不开、奶瓶数量不足、地址信息不正确或者其他原因，程序将会记录该操作，将本地数据库中的记录状态修改为相对应的状态。无论点击哪个按钮，都会在操作的同时记录更新的时间，将配送情况更新到本地数据库。

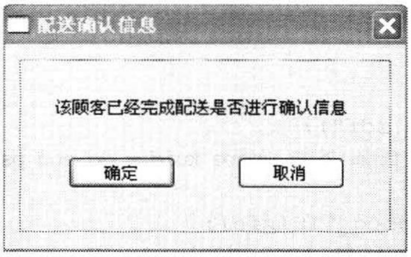



图8-17 配送确认信息界面

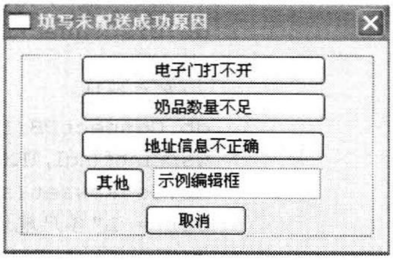



图8-18 填写未配送成功原因界面

### 2. 查询已送

图8-19显示了详细的已配送数据，用户可以通过点击方式的查询到指定的信息。

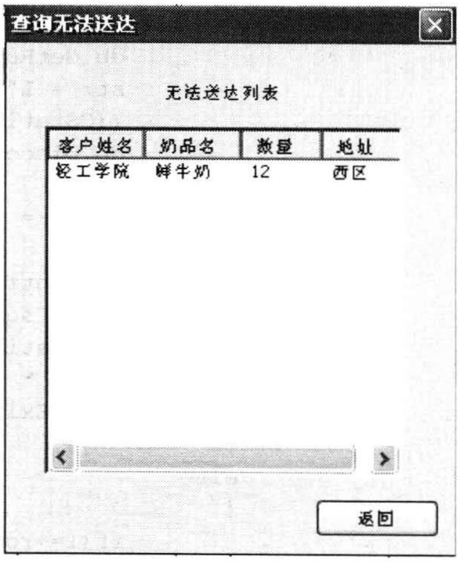



图8-19 已配送列表

该功能模块代码如下：

```
BOOL CQuerySendedDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    m_sendedList.InsertColumn(0,L"客户姓名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_sendedList.InsertColumn(1,L"奶品名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_sendedList.InsertColumn(2,L"数量",LVCFMT_
IMAGE|LVCFMT_LEFT,50);
    m_sendedList.InsertColumn(3,L"地址",LVCFMT_
IMAGE|LVCFMT_LEFT,60);

    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");

    CString str;
    TCHAR sql[1024];

    //客户ID
    wsprintf(sql,TEXT("select count(1)from PSRB"));
    HRESULT hr = db.GetRowset(sql);

    int rownumber = *db.g_TTSInfo - 48;//db.gTTSInfo是char值即ASCII码,转换为int型需要-48。即总条数
    db.DisconnectDB();
```

```
    int i=0;
    int kuid = 1;
    do
    {
        db.ConnectDB(L"MilkDeliver.sdf");
        wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d and psbz = 1"),kuid);
        HRESULT hr = db.GetRowset(sql);
        if(0 != _wcsicmp(L"",db.g_TTSInfo))
        {
            db.DisconnectDB();
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d"),kuid);
            db.GetRowset(sql);
            str = db.g_TTSInfo;
            m_sendedList.InsertItem(i,str);
            db.DisconnectDB();
            //////////////////
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d"),kuid);
            db.GetRowset(sql);
            str = db.g_TTSInfo;
            m_sendedList.SetItemText(i,1,str);
            db.DisconnectDB();

            //////////////////
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d"),kuid);
            db.GetRowset(sql);
            str = db.g_TTSInfo;
            m_sendedList.SetItemText(i,2,str);
            db.DisconnectDB();

            //////////////////
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d"),kuid);
            db.GetRowset(sql);
            str = db.g_TTSInfo;
            m_sendedList.SetItemText(i,3,str);
            db.DisconnectDB();

            i++;
        }
        else
            db.DisconnectDB();
        kuid++;
    }
    while(kuid<=rownumber);

    return TRUE;  // 除非将焦点设置到控件,否则返回TRUE
}
```

图8-18中显示了已配送的数据，如果想要查询更为详细的内容，用户可以直接点击指定的记录，程序将会显示详细已配送数据的内容。

### 3. 查询无法送达

图8-20显示了详细的无法送达数据，用户可以通过点击的方式查询到指定的信息。

该功能模块代码如下：

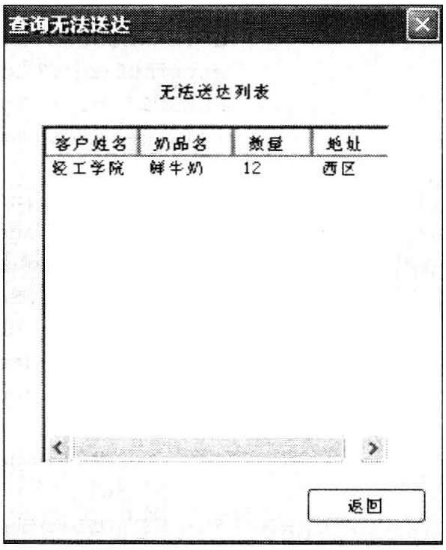



图8-20 无法送达列表

```
BOOL CQueryCantsendDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    m_cantsendList.InsertColumn(0,L"客户姓名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_cantsendList.InsertColumn(1,L"奶品名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_cantsendList.InsertColumn(2,L"数量",LVCFMT_
IMAGE|LVCFMT_LEFT,50);
    m_cantsendList.InsertColumn(3,L"地址",LVCFMT_
IMAGE|LVCFMT_LEFT,60);

    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");

    CString str;
    TCHAR sql[1024];

    //客户ID
    wsprintf(sql,TEXT("select count(1)from PSRB"));
    HRESULT hr = db.GetRowset(sql);

    int rownumber = *db.g_TTSInfo - 48;//db.gTTSInfo是char值即ASCII码,转换为int型需要-48。即总条数
    db.DisconnectDB();
    int i=0;
    int kuid = 1;
    do
    {
        db.ConnectDB(L"MilkDeliver.sdf");
        wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d and psbz = 2"),kuid);
        HRESULT hr = db.GetRowset(sql);
        if(0 != _wcsicmp(L"",db.g_TTSInfo))
        {
            db.DisconnectDB();
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d"),kuid);
            db.GetRowset(sql);
            str = db.g_TTSInfo;
            m_cantsendList.InsertItem(i,str);
            db.DisconnectDB();
            //////////////////
            db.ConnectDB(L"MilkDeliver.sdf");
            wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d"),kuid);
            db.GetRowset(sql);
            str = db.g_TTSInfo;
            m_cantsendList.SetItemText(i,1,str);
            db.DisconnectDB();
```

```
                    ///////////////////
                    db.ConnectDB(L"MilkDeliver.sdf");
                    wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d"),kuid);
                    db.GetRowset(sql);
                    str = db.g_TTSInfo;
                    m_cantsendList.SetItemText(i,2,str);
                    db.DisconnectDB();

                    ///////////////////
                    db.ConnectDB(L"MilkDeliver.sdf");
                    wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d"),kuid);
                    db.GetRowset(sql);
                    str = db.g_TTSInfo;
                    m_cantsendList.SetItemText(i,3,str);
                    db.DisconnectDB();

                    i++;
            }
            else
                    db.DisconnectDB();
            kuid++;
    }
    while(kuid<=rownumber);
    return TRUE;  // 除非将焦点设置到控件,否则返回TRUE
}
```

### 4. 查询全部

图8-21显示了详细的全部配送数据，用户可以通过点击方式查询到指定的信息。

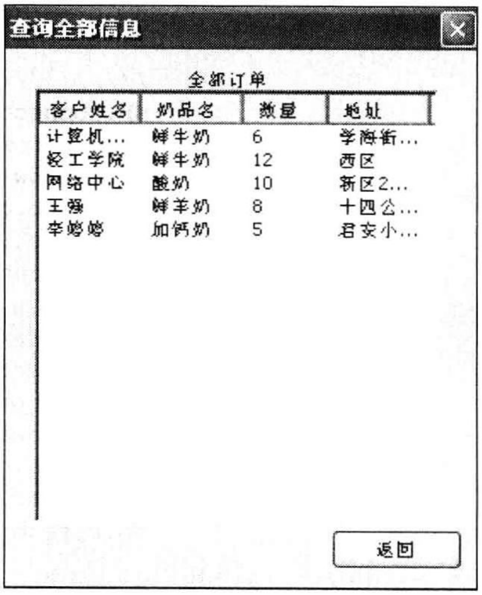



图8-21　全部列表

该功能模块代码如下：

```
BOOL CQueryAllDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    m_allList.InsertColumn(0,L"客户姓名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_allList.InsertColumn(1,L"奶品名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_allList.InsertColumn(2,L"数量",LVCFMT_
IMAGE|LVCFMT_LEFT,50);
    m_allList.InsertColumn(3,L"地址",LVCFMT_
IMAGE|LVCFMT_LEFT,60);

    Database db;
    db.ConnectDB(L"MilkDeliver.sdf");

    CString str;
    TCHAR sql[1024];

    //客户ID
```

```
wsprintf(sql,TEXT("select count(1)from PSRB"));
HRESULT hr = db.GetRowset(sql);

int rownumber = *db.g_TTSInfo - 48;//db.gTTSInfo是char值即ASCII码,转换为int型需要-48。即总条数
db.DisconnectDB();

int i=0;
int kuid = 1;
do
{
        db.ConnectDB(L"MilkDeliver.sdf");
        wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d"),kuid);
        HRESULT hr = db.GetRowset(sql);
        if(0 != _wcsicmp(L"",db.g_TTSInfo))
        {
                db.DisconnectDB();
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_allList.InsertItem(i,str);
                db.DisconnectDB();
                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_allList.SetItemText(i,1,str);
                db.DisconnectDB();

                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_allList.SetItemText(i,2,str);
                db.DisconnectDB();

                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_allList.SetItemText(i,3,str);
                db.DisconnectDB();

                i++;
        }
        else
                db.DisconnectDB();
        kuid++;
}
```

```
    while(kuid<=rownumber);
    return TRUE;  // 除非将焦点设置到控件,否则返回TRUE
}
```

## 8.5.4 数据传输模块

传输数据模块是配送每日与服务器进行数据更新的功能模块，可从服务器下载当日需配送的数据信息和上传配送结束后的成果信息。图8-22显示了订单下载与上传的界面效果。

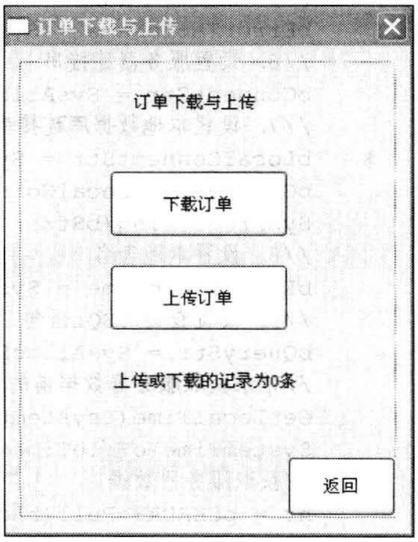



图8-22 订单上传与下载

下载订单代码如下：

```
void CUpdateDlg::OnBnClickedBtnDownload()
{
    // TODO: Add your control notification handler code here
    // TODO: Add your control notification handler code here
    ISSCERDA*                  pCERDA = NULL;
    BSTR bStr = NULL;
    BSTR bConnectStr = NULL;
    BSTR bQueryStr = NULL;
    BSTR bLocalConnectStr = NULL;
    BSTR bLocalTableName = NULL;
    BSTR bErrorTableName  = NULL;
    RDA_TRACKOPTION trackOption = TRACKINGOFF;

    CString internetURL = L"http://192.168.1.47/sqlce/sqlcesa30.dll";
    CString internetUsername = L"";
    CString internetPassword = L"";
    CString tracking = L"";
    CString errorTableName = L"tblErrors";
    CString connectStr = L"provider=sqloledb; data source=wang; Initial
Catalog=dbtest; user id=sa; password=123456";
    CString localConnectStr = L"Provider=microsoft.sqlserver.oledb.ce.3.0;Data
Source=\\MilkDeliver.sdf";
    CString localTableName = L"Menu";
    CString queryStr = L"select * from Menu";


    SYSTEMTIME                 systemtime;          //系统时间
    double                     timeTotal;           //总共消耗的时间
    __int64                    timeStart = 0;       //开始操作的时间
    __int64                    timeStop;            //结束操作的时间
    WCHAR  wszBuffer[512];

    //得到pCERDA接口
    HRESULT hr = CoCreateInstance(CLSID_RemoteDataAccess,NULL,CLSCTX_INPROC_
SERVER,IID_ISSCERDA, (LPVOID *)&pCERDA);
    ASSERT(hr == S_OK);
    ASSERT(pCERDA != NULL);
    //1. 设置Internet URL
    pCERDA->put_InternetURL(SysAllocString(internetURL));
    //2. 设置Internet userName
```

```
    pCERDA->put_InternetLogin(SysAllocString(internetUsername));
    //3. 设置Internet password
    pCERDA->put_InternetPassword(SysAllocString(internetPassword));
    //4. 得到跟踪类型
    trackOption = TRACKINGOFF;
    //5. 设置错误表名
    bErrorTableName = SysAllocString(errorTableName);
    //6. 设置服务器连接串
    bConnectStr = SysAllocString(connectStr);
    //7. 设置本地数据库连接串
    bLocalConnectStr = SysAllocString(localConnectStr);
    pCERDA->put_LocalConnectionString(bLocalConnectStr);
    SysFreeString(bStr);
    //8. 设置本地表名
    bLocalTableName = SysAllocString(localTableName);
    //9. 设置查询的SQL语句
    bQueryStr = SysAllocString(queryStr);
    //记录获取服务器数据前的时间
    GetLocalTime(&systemtime);
    SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStart);
    //获取服务器数据
    hr = pCERDA->Pull(bLocalTableName,
            bQueryStr,
            bConnectStr,
            trackOption,
            bErrorTableName);
    SysFreeString(bLocalTableName);
    SysFreeString(bQueryStr);
    SysFreeString(bConnectStr);
    SysFreeString(bErrorTableName);
    if(SUCCEEDED(hr))
    {
            //记录获取服务器数据后的时间
            GetLocalTime(&systemtime);
            SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStop);
            //记录获取服务器数据的总时间
            timeTotal = ((double)(timeStop - timeStart) / 10000000);
            swprintf(wszBuffer,_T("从服务器获取数据成功\r\n\r\总共消耗= %8.3f秒"),timeTotal);
            AfxMessageBox(wszBuffer);
    }
    else
    {
            //显示获取服务器数据错误
            ISSCEErrors  *pISSCEErrors = NULL;
            if(SUCCEEDED(hr = pCERDA->get_ErrorRecords(&pISSCEErrors)))
            {
                    ShowErrors(pISSCEErrors);
                    pISSCEErrors->Release();
            }
    }
    //释放远程访问接口
    pCERDA->Release();
}
```

上传订单数据代码如下：

```
void CUpdateDlg::OnBnClickedBtnUp()
{
    // TODO: Add your control notification handler code here
    ISSCERDA*                    pCERDA = NULL;
    BSTR bStr = NULL;
    BSTR bConnectStr = NULL;
    BSTR bLocalConnectStr = NULL;
    BSTR bLocalTableName = NULL;
    SYSTEMTIME                   systemtime;           //系统时间
    double                       timeTotal;            //总共消耗的时间
    __int64                      timeStart = 0;        //开始操作的时间
    __int64                      timeStop;             //结束操作的时间
    WCHAR   wszBuffer[512];
    CString internetURL = L"http://192.168.1.47/sqlce/sqlcesa30.dll";
    CString internetUsername = L"";
    CString internetPassword = L"";
    CString tracking = L"";
    CString errorTableName = L"tblErrors";
    CString connectStr = L"provider=sqloledb; data source=wang; Initial
Catalog=dbtest; user id=sa; password=123456";
    CString localConnectStr = L"Provider=microsoft.sqlserver.oledb.ce.3.0;Data
Source=\\MilkDeliver.sdf";
    CString localTableName = L"PSRB";
    CString queryStr = L"select * from PSRB";

    //得到pCERDA接口
    HRESULT hr = CoCreateInstance(CLSID_RemoteDataAccess,NULL,CLSCTX_INPROC_
SERVER,IID_ISSCERDA, (LPVOID *)&pCERDA);
    ASSERT(hr == S_OK);
    ASSERT(pCERDA != NULL);
    //1. 设置Internet URL
    bStr = SysAllocString(internetURL);
    pCERDA->put_InternetURL(bStr);
    SysFreeString(bStr);
    //2. 设置Internet userName
    bStr = SysAllocString(internetUsername);
    pCERDA->put_InternetLogin(bStr);
    SysFreeString(bStr);
    //3. 设置Internet password
    bStr = SysAllocString(internetPassword);
    pCERDA->put_InternetPassword(bStr);
    SysFreeString(bStr);
    //4. 设置服务器连接串
    bConnectStr = SysAllocString(connectStr);
    //5. 设置本地数据库连接串
    bLocalConnectStr = SysAllocString(localConnectStr);
    pCERDA->put_LocalConnectionString(bLocalConnectStr);
    SysFreeString(bStr);
    //6. 设置本地表名
    bLocalTableName = SysAllocString(localTableName);
    //记录向服务器提交数据前的时间
```

```
        GetLocalTime(&systemtime);
        SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStart);
        //向服务器提交数据
        hr = pCERDA->Push(bLocalTableName,bConnectStr,BATCHINGOFF);
        SysFreeString(bLocalTableName);
        SysFreeString(bConnectStr);
        if(SUCCEEDED(hr))
        {
                //记录向服务器提交数据后的时间
                GetLocalTime(&systemtime);
                SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStop);
                //记录向服务器提交数据总的时间
                timeTotal = ((double)(timeStop - timeStart) / 10000000);
                swprintf(wszBuffer,_T("向服务器提交数据成功\r\n\r\总共消耗= %8.3f秒"),timeTotal);
                AfxMessageBox(wszBuffer);
        }
        else
        {
                //显示向服务器提交数据失败信息
                ISSCEErrors  *pISSCEErrors = NULL;
                if(SUCCEEDED(hr = pCERDA->get_ErrorRecords(&pISSCEErrors)))
                {
                        ShowErrors(pISSCEErrors);
                        pISSCEErrors->Release();
                }
        }
        //释放远程访问接口
        pCERDA->Release();
    }
```

## 8.5.5 历史数据查询

历史数据查询模块是配送员对往日配送信息的查询模块，可对往日已配送或未配送的信息进行查询，不具备修改功能。图8-23显示了历史数据查询界面效果。

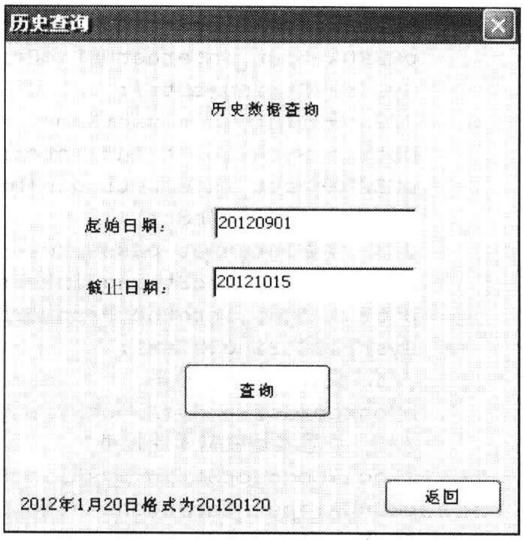



图8-23 历史数据查询

该功能模块代码如下：

```
BOOL CHistoryDeDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    m_hisList.InsertColumn(0,L"客户姓名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_hisList.InsertColumn(1,L"奶品名",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
    m_hisList.InsertColumn(2,L"数量",LVCFMT_
IMAGE|LVCFMT_LEFT,50);
    m_hisList.InsertColumn(3,L"地址",LVCFMT_
IMAGE|LVCFMT_LEFT,60);
```

```
Database db;
db.ConnectDB(L"MilkDeliver.sdf");

CString str;
TCHAR sql[1024];

//客户ID
wsprintf(sql,TEXT("select count(1)from PSRB"));
HRESULT hr = db.GetRowset(sql);

int rownumber = *db.g_TTSInfo - 48;//db.gTTSInfo是char值即ASCII码,转换为int型需要-48。即总条数
db.DisconnectDB();

int i=0;
int kuid = 1;
do
{
        db.ConnectDB(L"MilkDeliver.sdf");
        wsprintf(sql,TEXT("select kuid from PSRB where kuid = %d and psrq<'%s'
            and psrq>'%s'"),kuid,endvalue,beginvalue);
        HRESULT hr = db.GetRowset(sql);
        if(0 != _wcsicmp(L"",db.g_TTSInfo))
        {
                db.DisconnectDB();
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select kumc from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_hisList.InsertItem(i,str);
                db.DisconnectDB();
                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select npmc from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_hisList.SetItemText(i,1,str);
                db.DisconnectDB();

                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select pssl from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_hisList.SetItemText(i,2,str);
                db.DisconnectDB();

                //////////////////
                db.ConnectDB(L"MilkDeliver.sdf");
                wsprintf(sql,TEXT("select khdz from PSRB where kuid = %d"),kuid);
                db.GetRowset(sql);
                str = db.g_TTSInfo;
                m_hisList.SetItemText(i,3,str);
```

```
                    db.DisconnectDB();
                    i++;
            }
            else
                    db.DisconnectDB();
            kuid++;
    }
    while(kuid<=rownumber);
    return TRUE;  // 除非将焦点设置到控件,否则返回TRUE
}
```

## 8.5.6 配送员设置

配送员设置模块是配送员对终端进行设置的模块，可进行配送员编号和密码的设置，这一模块也提供服务器IP的设置，是终端与服务器通信的信息模块。图8-24显示了服务器及配送员信息设置界面效果。

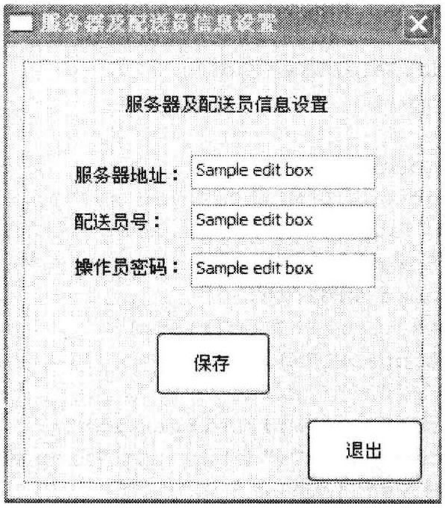



图8-24 服务器及配送员信息设置

至此，鲜奶配送系统各模块设计完毕。

# 第9章　无线点餐系统

无线点餐系统是在无线网络技术（目前主要是802.11b/802.11g无线局域网技术）平台基础上，针对餐饮行业所研发的综合性餐饮信息化解决方案软件。其最主要的特征实现随时随地的点菜。在信息化高度发展的今天，无线点餐管理系统凭借其完善餐饮行业的业务流程、高效方便的操作、稳定透明的财务查询、科学有力的决策依据、优雅温馨的环境档次提升，赢得了广大客户的认可。目前，国内餐饮信息化的发展正处在萌芽发展初期，由于餐饮业经营管理者计算机水平限制、对餐饮系统的理解不足以及根深蒂固的传统餐饮经营模式等因素的制约，使得餐饮行业信息化的普及严重滞后于其他行业的信息化进程。餐饮业行业利润增长所依赖无线点餐软件的份额也相对较小，不过值得乐观的一点是，国内餐饮管理系统的发展以及餐饮业对于无线点餐系统的选择使用呈现良好态势。21世纪餐饮行业的竞争是残酷的，不突破发展的瓶颈的餐饮企业难以适应未来的市场竞争，无线点餐管理系统可以为餐饮信息化的发展贡献力量。

传统的餐饮行业，点餐过程都是由餐厅服务员人工完成的。通常的过程是这样的，客人进入餐厅后，选择好桌位并开始点餐，服务员等待客人点餐完成，将客人的点餐内容送到厨房，由厨师下厨做菜。这一过程对于一个很小的餐馆来说没有问题，但是如果餐馆比较大或客人比较多，客人的等待时间就会比较长。为了解决这个问题，无线点餐系统应运而生，该系统一般由无线终端（一般是一个PDA）、无线路由器和后台服务器组成。这种技术架构很成熟，但是存在一个弊端就是PDA的价格较高，并且PDA只能作为点餐使用，这无形中增加了系统成本，也造成了资源的浪费。如果在Windows CE（简称WinCE）手机嵌入式系统中开发一套客户端软件来完成点餐功能，既避免了资源的浪费，又能够提高大型餐饮企业的工作效率。

## 9.1　搭建开发环境

无线点餐系统结构如图9-1所示。

本程序在Visual Studio 2005开发环境中编写与编译，使用C++与MFC作为开发的语言环境。分别在服务器端与WinCE终端安装SQL Server CE数据库环境。利用SQL Server CE数据库进行数据的存储与交换。

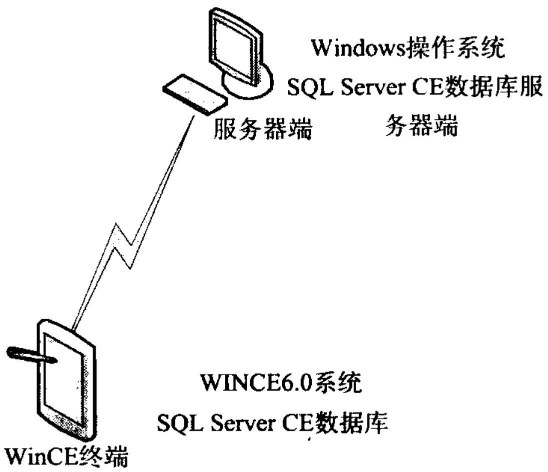



图9-1 无线点餐系统结构

### 9.1.1 服务器端SQL Server 2005数据库的安装

在服务器端安装SQL Server 2005，设置用户名为sa，密码为123456。

### 9.1.2 SQLCE数据库的安装与配置

SQL CE包括两个层面的安装，一是客户端安装，即在嵌入式或移动设备中安装SQLCE 3.0环境；二是服务器端安装，即在服务器端实现服务器同客户端设备的SQLCE数据同步、订阅等工作。如果只是在客户端应用，而不需要和服务器进行数据同步等工作，那么可以不用安装配置服务器环境。

#### 1. SQLCE客户端安装

SQLCE 3.0的客户端安装过程如下。

SQLCE 3.0安装文件可以从微软网站进行下载，也可以从Visual Studio 2005安装之后的目录中获取。在安装Visual Studio 2005时，会包含SQLCE 3.0安装文件，默认目录为C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\wce500\armv4i。该目录下包含了9个CAB安装文件。这9个安装文件对应3组；一组是针对smartphone智能手机的；一组是针对pocket pc智能PDA的；一组是针对WinCE 5.0通用系统的。

将sqlce30.ppc.wce5.armv4i.CAB、sqlce30.repl.ppc.wce5.armv4i.CAB和sqlce30.dev.CHS.ppc.wce5.armv4i.CAB这3个文件复制到PDA中，并依次进行安装。

其中sqlce30.ppc.wce5.armv4i.CAB安装文件是SQLCE 3.0核心安装文件；sqlce30.repl.ppc.wce5.armv4i.CAB安装文件是服务于复制、订阅等相关服务；sqlce30.dev.CHS.ppc.wce5.armv4i.CAB是SQLCE查询分析器的安装程序，以便于用户对SQLCE进行管理和操作，该查询分析器的默认目录是C:\Program Files\SQL Mobile\CN\isqlw30.exe。

在WinCE 6.0系统中安装这3个文件时，可能会提示这3个文件安装有可能不兼容，是否继

续安装。我们选择继续安装，因为这种情况不影响SQLCE 3.0的使用。安装完毕后，用户可以通过打开SQLCE查询分析器来建立SQLCE数据库以及相关数据表，并实现数据存储和查询等相关功能。

#### 2. SQLCE 3.0服务器端的安装与配置

服务器可以是普通PC，也可以是服务器。操作系统可以是Windows XP sp2、Windows 2000 Server、Windows Server 2003等。

若SQLCE 3.0应用程序要求在Microsoft SQL Server数据库和SQLCE数据库之间交换数据，则必须在服务器上安装如下软件：

- Microsoft Internet information Server（IIS）。
- SQLServer 2005或SQL Server 2000。
- SQLCE 3.0服务器工具。该工具的默认目录位于C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0下的Sqlce30setupcn.msi。安装后，系统中将会有“配置WEB同步向导”工具。

### 9.1.3 SQLCE 3.0连接配置

本节就以示例的形式介绍如何在客户端和服务器端建立代理连接，以便客户端能够和服务器端进行数据交互。

1）选择“开始”→“Microsoft SQL Server 2005 Mobile Edition”→“配置Web同步向导”项，打开SQLCE连接配置管理窗口，如图9-2所示。

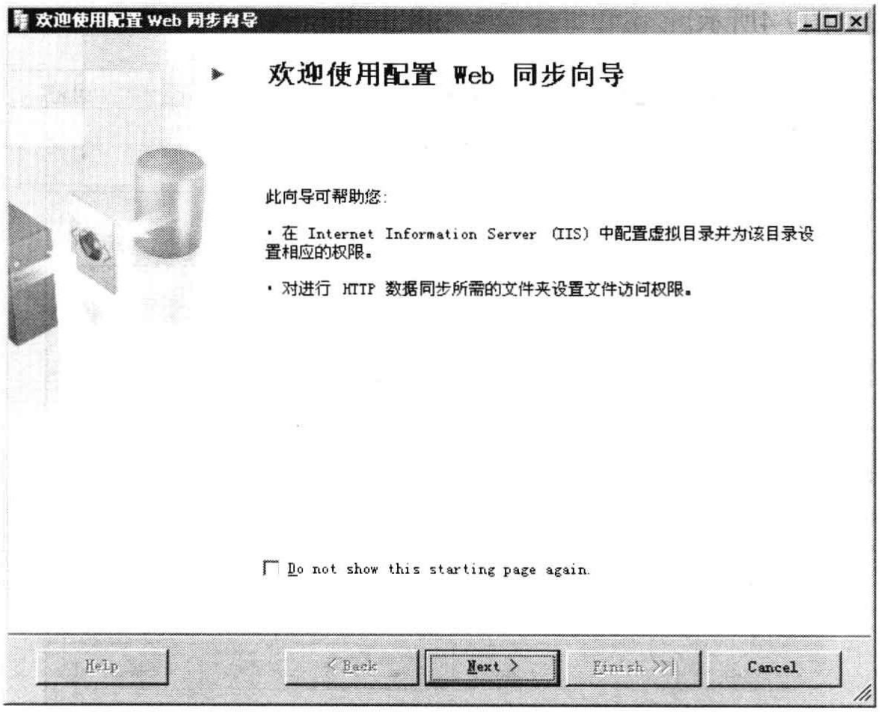



图9-2 SQLCE连接配置管理窗口

2）单击“Next”按钮，进入如图9-3所示的订阅服务器类型窗口，这里选择SQL Server Mobile Edition单选按钮。

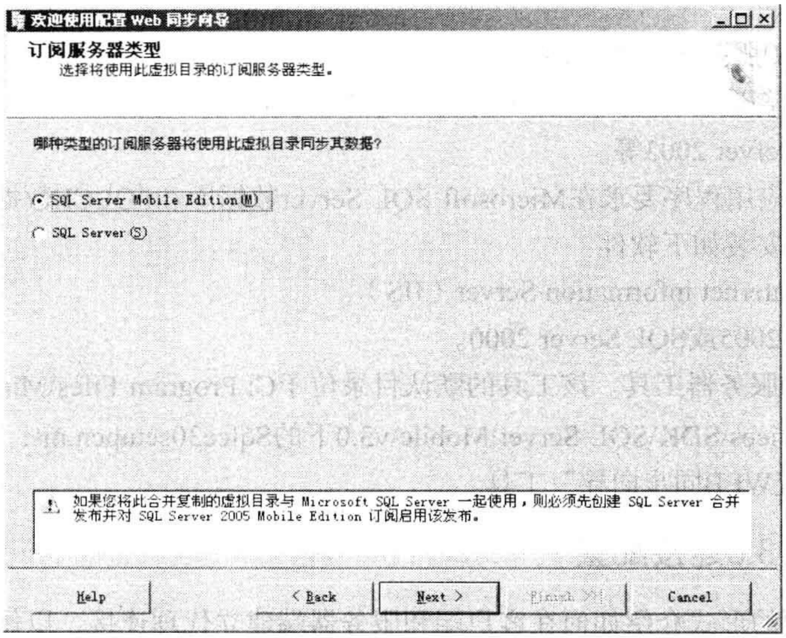



图9-3　订阅服务器类型

3）单击“Next”按钮，进入设置Web服务器窗口，这里输入运行IIS的计算机名，该示例中输入本计算机的网络名（默认即可），选择创建新的虚拟目录单选按钮，选中“默认网站”，具体设置如图9-4所示。

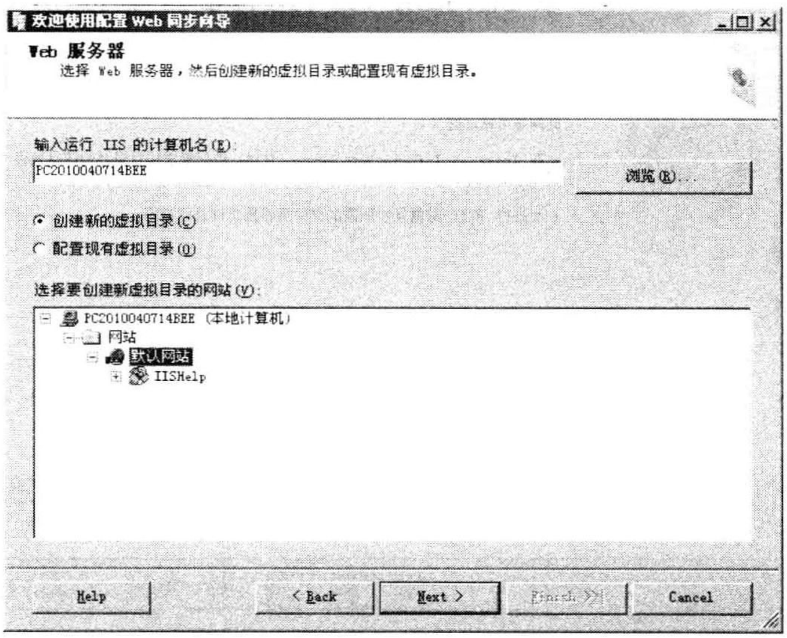



图9-4　设置Web服务器

4）单击“Next”按钮，进入如图9-5所示的设置虚拟目录信息窗口，这里将别名设置为sqlce，路径设置为默认。这个默认路径包含了sqlcesa30.dll文件。

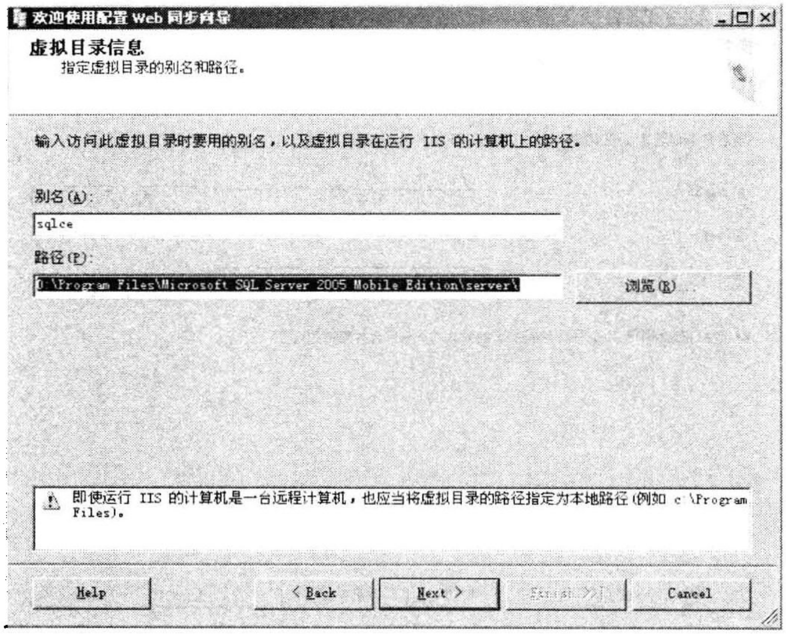



图9-5 虚拟目录信息

5）单击“Next”按钮，进入设置“安全通信”窗口，选择默认项。

单击“Next”按钮，进入设置客户端身份验证窗口，该示例中设置为匿名方式，如图9-6所示。

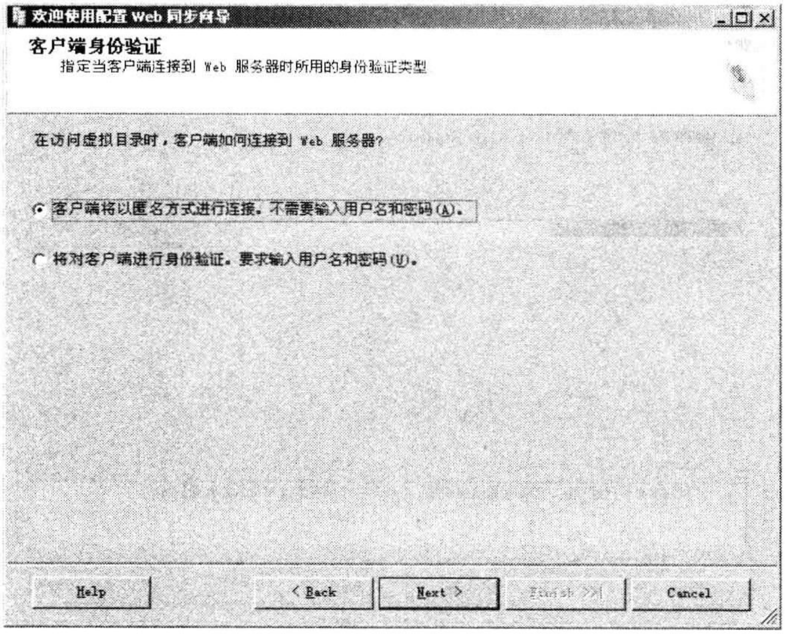



图9-6 客户端身份验证

6）单击“Next”按钮，进入匿名访问窗口设置“匿名访问”参数，默认即可，如图9-7所示。

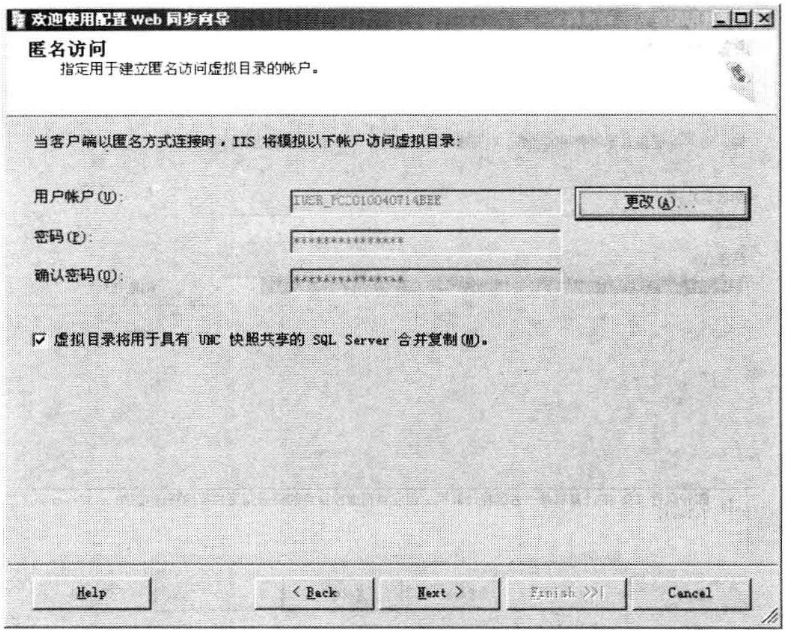



图9-7 设置匿名访问参数

7）单击“Next”按钮，进入设置快照共享访问窗口，这里需要一个共享文件目录。本示例在C盘建立了一个新目录snapshot，并将其设置成共享，具备读写权限，共享名为snapshot，因此在共享目录中输入\\计算机名\snapshot，如图9-8所示。

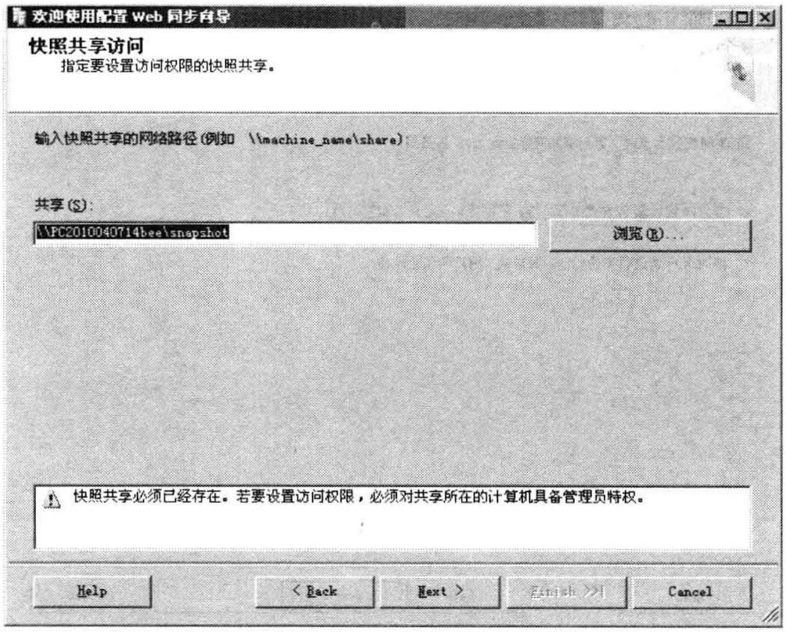



图9-8 设置快照共享访问

8）单击“Next”按钮便可以完成配置，配置状态如图9-9所示。

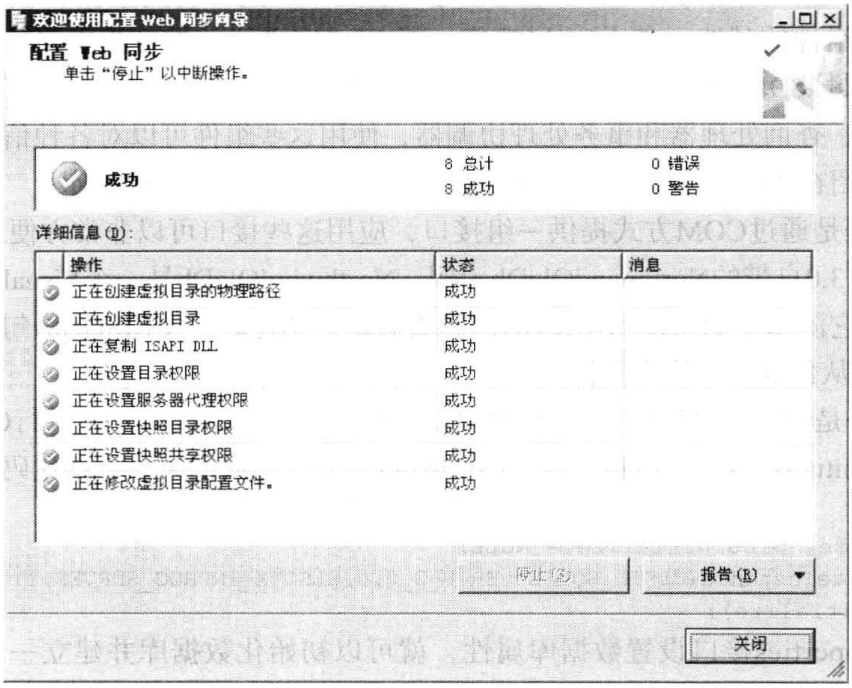



图9-9 配置状态

在SQLCE 3.0连接配置结束后，通过一个简单的方法可以判断配置是否成功，即在浏览器中输入SQL Server CE Server Agent（sqlcesa30.dll）的网络路径，当出现如图9-10所示的界面时，表示配置成功。在该例中B1-5为计算机名，sqlce为虚拟目录别名。

9）如果用户使用的是Windows XP SP2运行IIS，必须启用HTTP访问。系统默认设置为禁用HTTP访问。启动HTTP访问的操作步骤如下：

打开“控制面板”→单击Windows防火墙图标→单击“高级”按钮→在“网络连接设置”区域中单击“设置”按钮→选中“Web服务器（HTTP）”复选框→单击“确定”按钮以确认更改。

图9-10 Mobile Server验证成功

## 9.1.4 OLEDB编程访问SQLCE3.0

OLEDB是微软开发的一种高性能的、基于COM的数据库技术。OLEDB和其他Microsoft数据库技术的不同之处在于其提供通用数据访问方式的不同。OLEDB支持开发能访问各种数据源的应用程序，无论是DBMS还是非DBMS数据源。OLEDB通过使用支持某一给定数据源相应的DBMS功能的COM接口，使得对数据源的通用访问成为可能。COM不仅在数据源之间而且还在

其他各种应用程序之间减少了不必要的服务重复，同时还使它们之间的交互操作达到最大限度。

OLEDB是一组COM接口。用户可以通过一组统一的接口访问数据，从而把一个数据库组织成一个合作组件的基地。基于COM规范，OLEDB定义了一批可扩展并且可维护的接口，这些接口代管并封装DBMS功能中一致、可重复使用的部分。这些接口定义了DBMS组件的边界，例如行容器、查询处理器和事务处理协调器，使用这些组件可以对各种信息源进行统一事务访问。下面介绍在WinCE中基于OLEDB的C++编程访问SQLCE3.0数据库。

OLEDB编程是通过COM方式提供一组接口，应用这些接口可以非常方便地访问SQLCE，可以参考SQLCE 3.0自带的NorthwindOleDb示例，NorthwindOleDb是一个Visual C++ for Devices示例应用程序，它演示了如何使用OLEDB创建数据库、创建表、使用SQL语句插入数据，然后使用IRowsetSeek从数据库中检索数据。

IDBInitialize是OLEDB编程用于初始化和释放数据源对象的接口。在进行OLEDB编程时，首先要获得IDBInitialize接口，该接口通过CoCreateInstance函数创建，示例代码如下：

```
HRESULT hr;
IDBInitialize *pIDBInitialize = NULL;
Hr = CoCreateInstance(CLSID_SQLSErvERCE_3_0,0,CLSCTX_INPROC_SERVER,IID_IDBInitialize,
(void**)&pIDBInitialize);
```

结合IDBProperties接口设置数据库属性，就可以初始化数据库并建立一个数据库对话对象，代码如下：

```
IDBProperties *pIDBProperties=NULL;
……
//得到IDBProperties接口
hr = pIDBInitialize->QueryInterface(IID_IDBProperties,(void**)&pIDBProperties);
//设置数据库初始化属性
hr = pIDBPRoperties->SetProperties(1,dbpropset);
//初始化数据库
hr = pIDBInitialize->Initialize();
//得到访问数据库的回话对象IDBCreateSession
hr = pIDBInitialize->QueryInterface(IID_IDBCreateSession,(void**)&pIDBCreateSession);
```

建立数据库会话对象后，即可对数据库进行增删查改等操作。

在本程序中，自定义了一组OLEDB数据库操作类集合，将OLEDB中各个复杂的操作步骤简化为相应的函数，以便更加快捷地调用数据库功能。

该OLEDB类名为DataBase。存储于文件Database.h与Database.cpp文件中。在该类中首先定义ConnectDB函数，用来连接数据库。ConnectDB函数代码如下：

```
//连接数据库
HRESULT Database::ConnectDB(LPTSTR lpszDBName)
{
    HRESULT          hr = NOERROR;
    DBPROPSET        dbpropset[1];  // Property Set used to initialize provider
    DBPROP           dbprop[1];     // property array used in property set to initialize
provider
    //创建SQL Server CE供应商
    hr = CoCreateInstance( CLSID_SQLSERVERCE_3_0,0,CLSCTX_INPROC_SERVER,IID_
IDBInitialize, (void**)&pIDBInitialize);
```

```
    //验证
    if(FAILED(hr))
            goto CleanExit;
    //验证返回值
    //初始化
    VariantInit(&dbprop[0].vValue);
    //Initialize a property with name of database
    dbprop[0].dwPropertyID  = DBPROP_INIT_DATASOURCE;
    dbprop[0].dwOptions = DBPROPOPTIONS_REQUIRED;
    dbprop[0].vValue.vt = VT_BSTR;
    dbprop[0].vValue.bstrVal = SysAllocString(lpszDBName);

    //验证
    if(NULL == dbprop[0].vValue.bstrVal)
    {
            //设置返回值
            hr = E_OUTOFMEMORY;
            goto CleanExit;
    }
    //初始化属性集
    dbpropset[0].guidPropertySet = DBPROPSET_DBINIT;
    dbpropset[0].rgProperties = dbprop;
    dbpropset[0].cProperties = sizeof(dbprop)/sizeof(dbprop[0]);
    //查询IDB属性接口
    hr = pIDBInitialize->QueryInterface(IID_IDBProperties,
            (void **)&pIDBProperties);
    //验证
    if(FAILED(hr))
            goto CleanExit;
    else
            pIDBInitialize->AddRef();
    //创建数据库
    hr = pIDBProperties->SetProperties(1,
            dbpropset);
    //验证
    if(FAILED(hr))
            goto CleanExit;
    pIDBInitialize->Initialize();

    //创建新的数据库会话对象
    hr = CreateDBSession();
CleanExit:
    //释放内存
    VariantClear(&dbprop[0].vValue);
    if(pIDBProperties)
    {
            pIDBProperties->Release();
    }
    if(pIDBInitialize)
    {
            pIDBInitialize->Release();
    }
    //以上命令中任何一个失败，只有执行以下命令
```

```
    if(FAILED(hr))
            //断开数据库/复位OLEDB变量
            DisconnectDB();
    return hr;
}
```

CreateDBSession函数代码如下：

```
HRESULT Database::CreateDBSession(void)//创建会话
{
    HRESULT hr = NOERROR;
    //查询IDB创建会话对象
    hr = pIDBInitialize->QueryInterface(IID_IDBCreateSession,
            (void **)&pIDBCreateSession);
    if(FAILED(hr))
            goto CleanExit;
    //创建新数据库会话对象
    hr = pIDBCreateSession->CreateSession(NULL,IID_IDBCreateCommand,(IUnknown**) &pIDBCrtCmd);
    if(FAILED(hr))
            goto CleanExit;
    //创建一个命令对象的指针
    hr = pIDBCrtCmd->CreateCommand(NULL,
            IID_ICommandText,
            (IUnknown**)&pICmdText);
CleanExit:
    return hr;
}
```

执行CreateDBSession后，建立数据库会话对象pIDBCreateSession后，ConnectDB函数完成，即可以对数据库进行各种增删查改操作。这就需要利用GetRowset函数创建以便进行数据库行集的查找。函数如下：

```
HRESULT Database::GetRowset(LPTSTR lpszQuery)
{
    HRESULT              hr = NOERROR;
    //设置SQL查询语句
    hr = pICmdText->SetCommandText(DBGUID_SQL,
            lpszQuery);
    if(FAILED(hr))
            goto CleanExit;
    //执行SQL查询语句
    hr = pICmdText->Execute(NULL,
            IID_IRowset,
            NULL,
            NULL,
            (IUnknown **)&pIRowset);   //建立IRowset对象pIRowset
    if(!FAILED(hr))
    {
            if(NULL != pIRowset)
                    //继续通过检索行集对象
                    ProcessRowset(pIRowset);
    }
CleanExit:
    if(NULL != pIRowset)
```

```
    {
            pIRowset->Release();
            pIRowset = NULL;
    }
    if(NULL != pICmdText)
    {
            pICmdText->Release();
            pICmdText = NULL;
    }
    if(FAILED(hr))
            //断开数据库/复位的OLEDB变量
            DisconnectDB();
    return hr;
}
```

其中DisconnectDB函数中，释放了在连接数据库过程中建立的各个接口。清理内存单元。

若执行的SQL语句需要返回数据，由ProcessRowset（pIRowset）函数进行具体的返回操作。该函数代码如下：

```
HRESULT Database::ProcessRowset(IRowset *pIRowset)
{
    HRESULT  hr = NOERROR;
    //FOR LISTVIEW USED ONLY
    LVITEM  pitem;
    ULONG  lColumn = 0;
    ULONG  lNumCols= 0;
    ULONG  lCount  = 0;
    ULONG  lNumRowsRetrieved = 0;//../
    ULONG  ConsumerBufColOffset = 0;
    IAccessor  *pIAccessor   = NULL;
    IColumnsInfo *pIColumnsInfo = NULL;
    DBCOLUMNINFO *pDBColumnInfo = NULL;
    DBBINDING  *pBindings  = NULL;
    HACCESSOR  hAccessor   = NULL;
    HROW  hRows[10];
    HROW  *pRows    = &hRows[0];
    BYTE  *pBuffer = NULL;
    WCHAR  *pStringsBuffer = NULL;
    //从行集对象中获得的IColumnInfo接口
    hr = pIRowset->QueryInterface(IID_IColumnsInfo,
            (void **)&pIColumnsInfo);
    //验证
    if(FAILED(hr))
    {
            //更新状态
            MessageBox(NULL,TEXT("Failed to query IColumnsInfo interface!"),TEXT("error"),MB_OK);
            //终止当前程序
            goto CleanExit;
    }
    else
            pIRowset->AddRef();
    //检索的列信息
    pIColumnsInfo->GetColumnInfo(&lNumCols,
```

```
        &pDBColumnInfo,
        &pStringsBuffer);
//释放的列信息接口
pIColumnsInfo->Release();

//建立一个DBBINDING阵列
pBindings = new DBBINDING[lNumCols];

//使用ColumnInfo结构，填写pBindings阵列
for(lCount=0; lCount<lNumCols; lCount++)
{
        pBindings[lCount].iOrdinal    = lCount+1;
        pBindings[lCount].obValue     = ConsumerBufColOffset;
        pBindings[lCount].pTypeInfo   = NULL;
        pBindings[lCount].pObject     = NULL;
        pBindings[lCount].pBindExt    = NULL;
        pBindings[lCount].dwPart      = DBPART_VALUE;
        pBindings[lCount].dwMemOwner  = DBMEMOWNER_CLIENTOWNED;
        pBindings[lCount].eParamIO    = DBPARAMIO_NOTPARAM;
        pBindings[lCount].cbMaxLen    = 200;
        pBindings[lCount].dwFlags     = 1;
        pBindings[lCount].wType       = DBTYPE_WSTR;
        pBindings[lCount].bPrecision  = pDBColumnInfo[lCount].bPrecision;
        pBindings[lCount].bScale      = pDBColumnInfo[lCount].bScale;
        //Compute the next buffer offset.
        ConsumerBufColOffset += 200;
}
//获取IAccessor接口
hr = pIRowset->QueryInterface(IID_IAccessor,
        (void **)&pIAccessor);
//验证
if(FAILED(hr))
{
        //更新状态
        MessageBox(NULL,TEXT("Failed to query IColumnsInfo interface!"),TEXT("error"),MB_OK);
        //终止当前程序
        goto CleanExit;
}
else
        pIRowset->AddRef();
//创建从绑定集合的访问(pBindings)
hr=pIAccessor->CreateAccessor(DBACCESSOR_ROWDATA,
        lNumCols,
        pBindings,
        0,
        &hAccessor,
        NULL);
//获得10行1组
hr=pIRowset->GetNextRows(NULL,0,1,&lNumRowsRetrieved,&pRows);
//对于连续的缓冲区分配空间
pBuffer = new BYTE[ConsumerBufColOffset];
//显示的行
```

```
    while(lNumRowsRetrieved > 0)
    {
            //对于每一行打印列的数据
            for(lCount=0; lCount<lNumRowsRetrieved; lCount++)
            {
                    //初始化
                    memset(pBuffer,0,ConsumerBufColOffset);
                    //获得行数据的值
                    pIRowset->GetData(hRows[lCount],hAccessor,pBuffer);
                    //-- 项目
                    pitem.mask = LVIF_TEXT| LVIF_IMAGE | LVIF_PARAM;
                    //LVM_GETITEMCOUNT这一消息检索列表视图控件中的项目数
                    pitem.pszText = (LPTSTR)&pBuffer[pBindings[0].obValue];
                    if(pitem.pszText != NULL)
                    {
                            _tcscat(g_TTSInfo,_T(""));
                            _tcscat(g_TTSInfo,pitem.pszText);
                    }
            }
            //释放行检索
            hr=pIRowset->ReleaseRows(lNumRowsRetrieved,hRows,NULL,NULL,NULL);
            if(FAILED(hr))
            {
                    //更新状态
                    MessageBox(NULL,TEXT("Failed to query IColumnsInfo interface!"),TEXT("error"),MB_OK);
                    //终止当前程序
                    goto CleanExit;
            }
            //获得10行一组
            hr=pIRowset->GetNextRows(NULL,0,1,&lNumRowsRetrieved,&pRows);
            if(FAILED(hr))
            {
                    //更新状态
                    MessageBox(NULL,TEXT("Failed to query IColumnsInfo interface!"),TEXT("error"),MB_OK);
                    //终止当前程序
                    goto CleanExit;
            }
    }
CleanExit:
    if(NULL != pColumnsInfo)
    {
            pIColumnsInfo->Release();
            pIColumnsInfo = NULL;
    }
    if(NULL != pBindings)
    {
            delete [] pBindings;
            pBindings = NULL;
    }
    if(NULL !=pBuffer)
    {
            delete [] pBuffer;
```

```
            pBuffer = NULL;
        }
        if(NULL != pIAccessor)
        {
            pIAccessor->Release();
            pIAccessor = NULL;
        }
        return hr;
    }
```

、以上介绍了我们在程序中将使用到的Database类以及类中的各成员函数，利用这些成员函数我们能够在WINCE环境中创建、查询或更改数据库表，完成程序设计中需要的各种功能。

## 9.2 初始化调用环境

在系统的初始化过程中，首先需要初始化COM环境。在OnInitDialog函数中添加如下代码：

```
CoInitializeEx(NULL,COINIT_MULTITHREADED);//CM
```

初始化后程序需要建立数据库表，如果数据库表已经建立，则打开数据表。代码如下：

```
void Database::Finddatabase(LPTSTR dbname)
{
    //TODO: Add your control notification handler code here
    HRESULT hr = NOERROR;           //错误代码的报告
    HANDLE hFind;                   //文件句柄
    WIN32_FIND_DATA FindFileData;   //文件结构说明

    //如果数据库存在,则打开
    //否则将样本数据创建一个新的数据库
    //
    hFind = FindFirstFile(dbname, &FindFileData);
    if(INVALID_HANDLE_VALUE == hFind)
    {
            DeleteFile(dbname);
//////////////////////Users_Table////////////////////////////
            LPTSTR DropSql_Users = L"DROP TABLE Users";
            LPTSTR CreateSql_Users = L"CREATE TABLE Users(UserID INT,UserName
NVARCHAR(20),UserKey NVARCHAR(10))";
            LPTSTR CreateIndexSql_Users = L"CREATE INDEX PK_Users on Users(UserID ASC)";
            //////////////////////Tables_Table////////////////////////////
            LPTSTR DropSql_Tables = L"DROP TABLE Tables";
            LPTSTR CreateSql_Tables = L"CREATE TABLE Tables(TableID INT,CumNum NVARCHAR(20))";
            LPTSTR CreateIndexSql_Tables  = L"CREATE INDEX PK_Tables on Tables(TableID ASC)";
            //////////////////////MenuType_Table////////////////////////////
            LPTSTR DropSql_MenuType = L"DROP TABLE MenuType";
            LPTSTR CreateSql_MenuType = L"CREATE TABLE MenuType(id INT,name NVARCHAR(20))";
            LPTSTR CreateIndexSql_MenuType = L"CREATE INDEX PK_MenuType on MenuType(id ASC)";
            //////////////////////Menu_Table////////////////////////////
            LPTSTR DropSql_Menu = L"DROP TABLE Menu";
            LPTSTR CreateSql_Menu = L"CREATE TABLE Menu(TypeId INT,name NVARCHAR(20),price
```

```
NVARCHAR(20))";
                LPTSTR CreateIndexSql_Menu = L"CREATE INDEX PK_Menu on Menu(TypeId ASC)";
                ///////////////////////Order_Table//////////////////////////////
                LPTSTR DropSql_Order = L"DROP TABLE Order";
                LPTSTR CreateSql_Order = L"CREATE TABLE OrderTbl(UserID int,OrderTime
NVARCHAR(20),TableID int,PersonNum int,IsPay int)";
                LPTSTR CreateIndexSql_Order = L"CREATE INDEX PK_OrderTbl on OrderTbl(UserID ASC)";
                ///////////////////////OrderDetail_Table//////////////////////////////
                LPTSTR DropSql_OrderDetail = L"DROP TABLE OrderDetail";
                LPTSTR CreateSql_OrderDetail = L"CREATE TABLE OrderDetail(OrderID int,MenuID
int,num int)";
                LPTSTR CreateIndexSql_OrderDetail = L"CREATE INDEX PK_OrderDetail on
OrderDetail(OrderID ASC)";
                // Create  database
                hr = CreateDatabase(dbname);
                hr = ConnectDB(dbname);
                hr = GetRowset(CreateSql_Users);
                hr = GetRowset(CreateSql_Tables);
                hr = GetRowset(CreateSql_MenuType);
                hr = GetRowset(CreateSql_Menu);
                hr = GetRowset(CreateSql_Order);
                hr = GetRowset(CreateSql_OrderDetail);
                DisconnectDB();
        }
    Exit:
        ;
    }
```

至此，数据库初始化完毕，并建立了pIDBCreateSession接口用于获取一个新的数据库访问会话对象。在访问数据库时，都要先获取一个数据库访问会话对象，然后基于此会话对象执行SQL语句，添加、删除数据以及事务的相关操作。

## 9.3 终端应用程序设计

### 9.3.1 系统的登录功能

为了增强系统的使用安全，使用系统之前必须登录系统，登录过程是通过网络，在后台的数据库通过用户名称和密码进行查询，注册用户方可使用该系统。系统登录界面如图9-11所示。

用户名与密码文本框ID分别为IDC_EDIT_USER与IDC_EDIT_KEY。

在函数中，利用IOpenRowset打开行集，并逐行进行搜索匹配用户名与密码，若匹配一致，则进入系统主菜单，否则提示密码错误或用户名不存在。

图9-11 系统登录界面

在点击“登录”按钮后，首先我们获取用户名与密码编辑框中的数值：设定登录按钮函数，获取用户输入的用户名与密码，并与Users数据表中数据进行比对，完成用户登录，登录按钮代码如下：

```
    HRESULT hr = NOERROR;
    Database db_log;
    hr = db_log.ConnectDB(L"WirelessOrder.sdf");
    CString tmp1,tmp2;
    GetDlgItem(IDC_EDIT_USERNAME)->GetWindowTextW(tmp1);
    const wchar_t* Username = tmp1;
    GetDlgItem(IDC_EDIT_USERKEY)->GetWindowTextW(tmp2);
    const wchar_t* Userkey = tmp2;
    TCHAR szBuffer[1024];
    wsprintf(szBuffer,TEXT("SELECT UserKey FROM Users WHERE UserName='%s'"),Username);
    hr = db_log.GetRowset(szBuffer);
    const wchar_t* TTSInfo = db_log.g_TTSInfo;
    if(0 == _wcsicmp((WCHAR*)L"",(WCHAR*)(TTSInfo)))
    {
            MessageBox(L"用户名不存在！");
            goto Exit;
    }
    else if(0 == _wcsicmp((WCHAR*)Userkey,(WCHAR*)(TTSInfo)))
    {
            CMainForm mainf = new CMainForm;
            mainf.DoModal();
    }
    else
    {
            MessageBox(L"密码错误！请重新输入！");
            goto Exit;
    }
Exit:
    db_log.DisconnectDB();
}
```

## 9.3.2 系统主菜单

系统登录成功之后便进入了系统主菜单，主菜单通过一个图形菜单的方式来展现系统的功能，单击某个选项便进入该功能的操作界面。运行界面如图9-12所示。

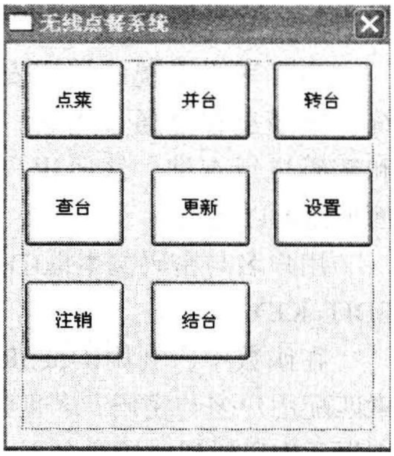
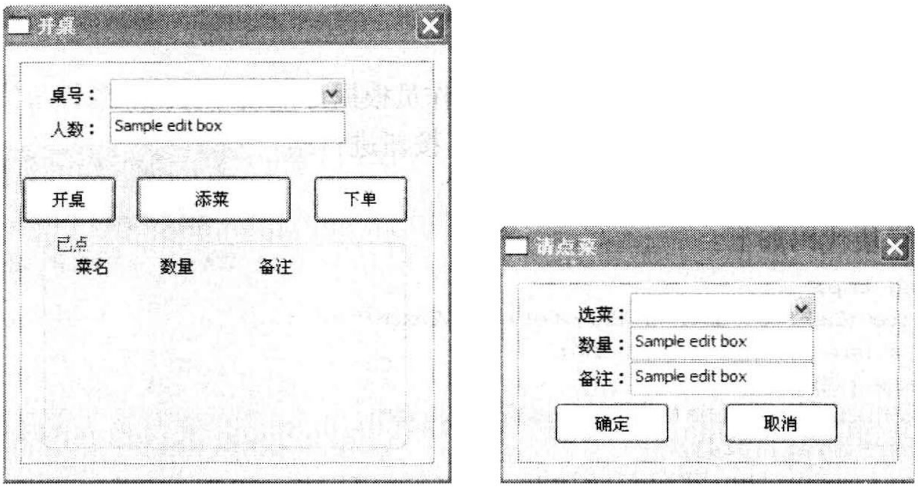



图9-12 系统主菜单

## 9.3.3 点餐功能

点餐功能模块是本系统中一个重要的功能模块，该模块要完成的任务是操作员输入顾客的点餐信息，通过网络及时地将点餐的信息传送到后台服务器，进而在厨房终端显示，有利于厨师尽快下厨做菜。

该功能包括了三部分：开桌、点菜和下单。运行界面如图9-13所示。




a）开桌　　b）点菜

图9-13　点餐运行界面

首先输入桌号与人数后单击“开桌”按钮生成订单。代码如下：

```
Database opendb;
opendb.ConnectDB(L"WirelessOrder.sdf");
TCHAR szBuffer[1024];
wsprintf(szBuffer,TEXT("insert into Tables values(%s,%s,%s,'%s')",tableID,num,flag));
opendb.GetRowset(szBuffer);
MessageBox(L"开桌成功");
opendb.DisconnectDB();
```

添菜界面代码为：

```
const wchar_t* foodID;
const wchar_t* orderID;
const wchar_t* foodnum;
const wchar_t* note;
Database opdb;
opdb.ConnectDB(L"WirelessOrder.sdf");
TCHAR szBuffer[1024];
wsprintf(szBuffer,TEXT("insert into OrderDetail values(%s,%s,%s,'%s')",orderID,
foodID,foodnum,note));
opdb.GetRowset(szBuffer);
MessageBox(L"添菜成功！");
opdb.DisconnectDB();
```

客人点完菜后单击“下单”按钮生成订单，代码如下：

```
Database opendb;
opendb.ConnectDB(L"WirelessOrder.sdf");
TCHAR szBuffer[1024];
wsprintf(szBuffer,TEXT("insert into Order values('%s',%s,%s,%s,%s,%s)",ordertime,
orderID,tableID,userID,persionnum,ispay));
hr = opendb.GetRowset(szBuffer);
MessageBox(L"下单成功");
opendb.DisconnectDB();
```

## 9.3.4 结算功能

顾客就餐结束需要结算，结算的过程是：操作员根据订单编号查询点餐订单信息和订单信息详细列表，费用得到顾客确认后，单击“结算”按钮进行结算。运行界面如图9-14所示。

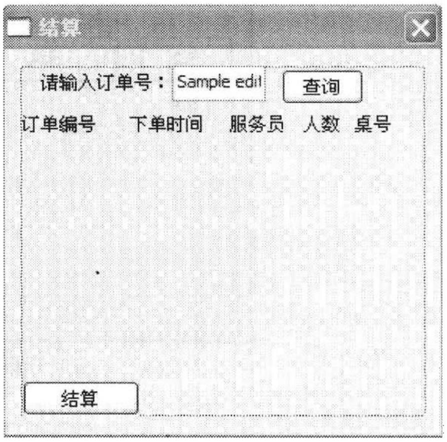



图9-14 结算功能模块运行界面

结算功能模块代码如下：

```
CString tmp;
GetDlgItem(CHECKOUT_EDIT_ORDER)->GetWindowTextW(tmp);
const wchar_t* orderID = tmp;
Database pCheckoutdb;
pCheckoutdb.ConnectDB(L"WirelessOrder.sdf");
TCHAR szBuffer[1024];
wsprintf(szBuffer,TEXT("select * from orders
where orderID = %s)",orderID));
pCheckoutdb.GetRowset(szBuffer);
Write2ListCtl(pCheckoutdb.g_TTSInfo);//函数
Write2ListCtl将SQL查询得到的行写入ListControl
MessageBox(L"结台成功! ");
pCheckoutdb.DisconnectDB();
```

## 9.3.5 查台功能

当有顾客进入餐馆时，需要服务员查询餐桌状态（有人或空位），来安排顾客就餐。这一过程是通过查询数据库中的餐桌数据得到当前餐桌状态列表，通过可视化的界面来进行显示。程序运行界面如图9-15所示。

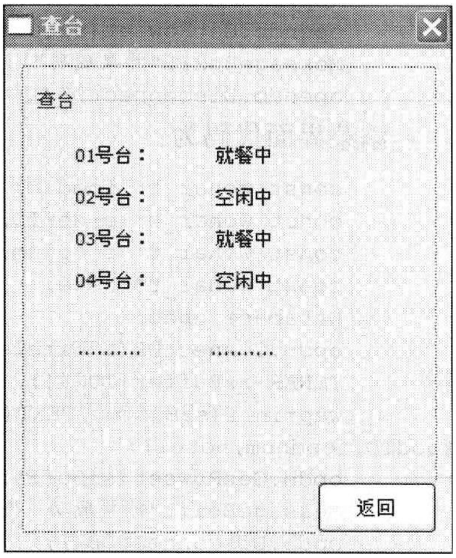



图9-15 查台功能模块运行界面

该模块即查询tables表中每桌的flag值，若flag值为0则空闲，若flag值为1则正在就餐中。添加GetRowSet()函数，在初始化时即读取数据库并输出桌号及状态。

```
Database chktldb;
int OUTFLAG[10][1024];
chktldb.ConnectDB(L"WirelessOrder.sdf");
for(int i=0;i<10,i++)
{
        TCHAR szBuffer;
        wsprintf(szBuffer,TEXT("select flag from
tables where tableID = %s"),i);
        chktldb.GetRowset(szBuffer);
        *OUTFLAG[i] = chktldb.g_TTSInfo;
}
chktldb.DisconnectDB();
```

## 9.3.6 转台功能

转台功能是指客户下单后，要求更换座位，这时需要更改已经提交的数据。更改的过程是根据订单号找到该订单对应的桌号，将该桌号更改为更换后的桌号。程序运行界面如图9-16所示。

转台功能模块代码如下：

图9-16 转台功能模块运行界面

```
    HRESULT hr = NOERROR;
    Database CTDlg_db;
    CString str;
    const wchar_t* orderid;
    CTDlg_db.ConnectDB(L"WirelessOrder.sdf");
    GetDlgItem(CHANGETABLE_EDIT_ORDERID)->GetWindowTextW(str);
    orderid = str;
    const wchar_t* newtableid;
    const wchar_t* oldtableid ;
    TCHAR szbuffer[1024];
    wsprintf(szbuffer,TEXT("select tableID from Order where orderID = %s",orderid));
    hr = CTDlg_db.GetRowset(szbuffer);
    str = CTDlg_db.g_TTSInfo;
    oldtableid = str;
    wsprintf(szbuffer,TEXT("updata Tables set flag = 0 where tableID = %s",oldtableid));
    hr = CTDlg_db.GetRowset(szbuffer);
    wsprintf(szbuffer,TEXT("updata Order set tableID = %s where orderID = %s",newtableid,orderid));
    hr = CTDlg_db.GetRowset(szbuffer);
    wsprintf(szbuffer,TEXT("updata Tables set flag = 1 where tableID = %s",newtableid));
    hr = CTDlg_db.GetRowset(szbuffer);
    CTDlg_db.DisconnectDB();
```

在该模块中，首先需要根据订单号获得当前桌的桌号，然后设置当前桌为空闲状态即flag=0，然后更新订单号对应的新桌号，最后将新桌号flag设为就餐状态（1）。

## 9.3.7 更新功能

为了提高程序的运行效率，将服务器中菜谱表（MenuTbl）和餐桌表（TableTbl）中的数据保存到客户端SQL CE数据库中。因此，当服务器中的数据改变时需要及时将服务器中的数据动态地更新到客户端，更新功能就是为此而设计的。程序运行界面如图9-17所示。

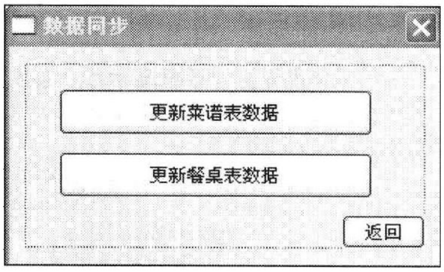


图9-17 更新功能模块运行界面

该模块需要利用无线网络远程访问服务器数据库，这里利用RDA技术实现数据库间的同步与更新，分别对菜谱表与餐桌表进行更新。

程序需要用到RDA定义的接口类，因此要在该CUpdateDlg类中引用ca_mergex30.h文件，代码如下：

```
#include <ca_merge30.h>
```

在执行远程数据访问过程中，可能会出现各种原因引起的错误，SQLCE提供了ISSCEError接口，ISSCEError接口、ISSCEParams接口和ISSCEParam接口来获取发生错误的详细信息。首先定义ShowError函数。该函数使用ISSCEErrors接口封装显示RDA错误方法。通过弹出对话框的方式来提示错误信息。该函数定义如下：

```
//显示远程数据访问时的错误
void CUpdateDlg::ShowErrors(ISSCEErrors* pISSCEErrors)
{
    HRESULT         hr;
    LONG            cbBuf;
    LONG            i;
    LONG            lErrorCount;
    LONG            lErrorIndex;
    LONG            lParamCount;
    LONG            lParamIndex;
    VARIANT         var;
    VARIANT         varParam;
    WCHAR           wszBuff[4096];
    WCHAR*          pwszBuffPos = &wszBuff[0];
    BSTR            bstr;
    ISSCEError*     pISSCEError = NULL;
    ISSCEParams*    pISSCEParams = NULL;
    ISSCEParam*     pISSCEParam = NULL;
    BOOL            fSuccess = FALSE;

    //初始化变量
    VariantInit(&var);
    VariantInit(&varParam);

    //通过IISSCEErrors接口得到发生错误的数量
    if(FAILED(hr = pISSCEErrors->get_Count(&lErrorCount)))
            goto Exit;
    if(lErrorCount <= 0)
    {
            ::MessageBox(NULL,L"没有错误信息",L"显示错误",MB_OK);
            fSuccess = TRUE;
            goto Exit;
    }

    //用弹出对话框显示每一条发生的错误
    for(lErrorIndex = 0; lErrorIndex < lErrorCount; lErrorIndex++)
    {
            cbBuf = swprintf(pwszBuffPos,L"ERROR  %d of %d\r\n",
                    lErrorIndex+1,lErrorCount);
            pwszBuffPos += cbBuf;
            //得到错误记录
            var.vt = VT_I4;
            var.lVal = lErrorIndex;
            if(FAILED(hr = pISSCEErrors->get_Item(var, &pISSCEError)))goto Exit;
            //得到错误详细信息
            if(FAILED(hr = pISSCEError->get_Description(&bstr)))goto Exit;
            cbBuf = swprintf(pwszBuffPos,L"DESCRIPTION: '%s'\r\n",bstr);
            pwszBuffPos += cbBuf;
            SysFreeString(bstr);
            //错误信息编号
            if(FAILED(hr = pISSCEError->get_Number(&i)))goto Exit;
            cbBuf = swprintf(pwszBuffPos,L"NUMBER: %8.8X\r\n",i);
            pwszBuffPos += cbBuf;
            //本地错误
            if(FAILED(hr = pISSCEError->get_NativeError(&i)))goto Exit;
```

```
            cbBuf = swprintf(pwszBuffPos,L"NATIVE_ERROR: %d\r\n",i);
            pwszBuffPos += cbBuf;
            //得到错误源
            if(FAILED(hr = pISSCEError->get_Source(&bstr)))goto Exit;
            cbBuf = swprintf(pwszBuffPos,L"SOURCE: '%s'\r\n",bstr);
            pwszBuffPos += cbBuf;
            SysFreeString(bstr);
            //得到错误参数信息
            if(FAILED(hr = pISSCEError->get_Params(&pISSCEParams)))goto Exit;
            //得到错误参数的个数
            if(FAILED(hr = pISSCEParams->get_Count(&lParamCount)))goto Exit;
            //显示每一个参数
            for(lParamIndex = 0; lParamIndex < lParamCount; lParamIndex++)
            {
                    //得到错误参数对象
                    var.vt = VT_I4;
                    var.lVal = lParamIndex;
                    if(FAILED(hr = pISSCEParams->get_Item(var, &pISSCEParam)))goto Exit;
                    //得到并显示错误参数值
                    if(FAILED(hr = pISSCEParam->get_Param(&varParam)))goto Exit;
                    if(VT_I4 == varParam.vt || VT_UI4 == varParam.vt)
                    {
                            cbBuf = swprintf(pwszBuffPos,L"P%d: %d\r\n",lParamIndex,
                                    (LONG)varParam.lVal);
                    }
                    else if(VT_I2 == varParam.vt || VT_UI2 == varParam.vt)
                    {
                            cbBuf = swprintf(pwszBuffPos,L"P%d: %d\r\n",lParamIndex,
                                    (LONG)varParam.iVal);
                    }
                    else if(VT_BSTR == varParam.vt)
                    {
                            cbBuf = swprintf(pwszBuffPos,L"P%d: '%s'\r\n",lParamIndex,
                                    varParam.bstrVal);
                    }
                    pwszBuffPos += cbBuf;
                    //清空参数变量
                    VariantClear(&varParam);
                    //释放参数对象
                    pISSCEParam->Release();
                    pISSCEParam = NULL;
            }
            cbBuf = swprintf(pwszBuffPos,L"\r\n");
            pwszBuffPos += cbBuf;

    }
    //用弹出框显示错误信息
    ::MessageBox(NULL,wszBuff,L"Error",MB_OK);
    fSuccess = TRUE;
Exit:
    //释放错误参数对象
    if(pISSCEParam)
    {
            pISSCEParam->Release();
            pISSCEParam = NULL;
```

```
}
//释放错误参数集对象
if(pISSCEParams)
{
        pISSCEParams->Release();
        pISSCEParams = NULL;
}

//释放错误对象
if(pISSCEError)
{
        pISSCEError->Release();
        pISSCEError = NULL;
}

//如果获取错误信息失败
if(!fSuccess)
{
        ::MessageBox(NULL,L"得到错误信息失败!",L"显示错误",MB_OK);
}
return;
```

在更新菜谱功能中，我们利用RDA技术中的Pull方法从服务器数据库提取数据至移动设备端数据库。由如下代码实现：

```
ISSCERDA*  pCERDA = NULL;
BSTR bStr = NULL;
BSTR bConnectStr = NULL;
BSTR bQueryStr = NULL;
BSTR bLocalConnectStr = NULL;
BSTR bLocalTableName = NULL;
BSTR bErrorTableName = NULL;
RDA_TRACKOPTION trackOption = TRACKINGOFF;

CString internetURL = L"http://192.168.1.47/sqlce/sqlcesa30.dll";
CString internetUsername = L"";
CString internetPassword = L"";
CString tracking = L"";
CString errorTableName = L"tblErrors";
CString connectStr = L"provider=sqloledb; data source=wang; Initial Catalog=dbtest; user id=sa; password=123456";
CString localConnectStr = L"Provider=microsoft.sqlserver.oledb.ce.3.0;Data Source=\\WirelessOrder.sdf";
CString localTableName = L"Menu";
CString queryStr = L"select * from Menu";

SYSTEMTIME          systemtime;         // 系统时间
double              timeTotal;          // 总共消耗的时间
__int64             timeStart = 0;      // 开始操作时的时间
__int64             timeStop;           // 结束操作时的时间
WCHAR   wszBuffer[512];

//得到pCERDA接口
HRESULT hr = CoCreateInstance(CLSID_RemoteDataAccess,NULL,CLSCTX_INPROC_SERVER,IID_ISSCERDA, (LPVOID *)&pCERDA);
ASSERT(hr == S_OK);
```

```
ASSERT(pCERDA != NULL);
//1. 设置Internet URL
pCERDA->put_InternetURL(SysAllocString(internetURL));

//2. 设置Internet userName
pCERDA->put_InternetLogin(SysAllocString(internetUsername));

//3. 设置Internet password
pCERDA->put_InternetPassword(SysAllocString(internetPassword));

//4. 得到跟踪类型
trackOption = TRACKINGOFF;


//5. 设置错误表名
bErrorTableName = SysAllocString(errorTableName);

//6. 设置服务器连接串
bConnectStr = SysAllocString(connectStr);

//7. 设置本地数据库连接串
bLocalConnectStr = SysAllocString(localConnectStr);
pCERDA->put_LocalConnectionString(bLocalConnectStr);
SysFreeString(bStr);

//8. 设置本地表名
bLocalTableName = SysAllocString(localTableName);

//9. 设置查询的SQL语句
bQueryStr = SysAllocString(queryStr);

//记录获取服务器数据前的时间
GetLocalTime(&systemtime);
SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStart);

//获取服务器数据
hr = pCERDA->Pull(bLocalTableName,
        bQueryStr,
        bConnectStr,
        trackOption,
        bErrorTableName);

SysFreeString(bLocalTableName);
SysFreeString(bQueryStr);
SysFreeString(bConnectStr);
SysFreeString(bErrorTableName);
.
if(SUCCEEDED(hr))
{
        //记录获取服务器数据后的时间
        GetLocalTime(&systemtime);
        SystemTimeToFileTime(&systemtime, (FILETIME*)&timeStop);
        //记录获取服务器数据的总时间
        timeTotal = ((double)(timeStop - timeStart) / 10000000);
        swprintf(wszBuffer,_T("从服务器获取数据成功\r\n\r\总共消耗= %8.3f秒"),timeTotal);
```

```
        AfxMessageBox(wszBuffer);
}
else
{
        //显示获取服务器数据错误
        ISSCEErrors  *pISSCEErrors = NULL;
        if(SUCCEEDED(hr = pCERDA->get_ErrorRecords(&pISSCEErrors)))
        {
                ShowErrors(pISSCEErrors);
                pISSCEErrors->Release();
        }
}
//释放远程访问接口
pCERDA->Release();
```

同样，在更新餐桌信息功能中，只需要将CString localTableName = L"Menu";中数据表Menu改为餐桌表Tables即可。即：

```
CString localTableName = L"Tables";
```

## 9.4 系统数据库设计

程序一般包括：输入、处理和输出。本系统也不例外，除了WinCE客户端应用程序和中间无线网络，还要有强大的数据库做支持。本系统WinCE客户端采用SQL CE 3.0数据库，服务器端采用SQL Server数据库，并应提供客户端数据和服务器端数据的同步功能，提高程序的运行效率。本节将对系统使用到的表结构进行详细规划作为设计的参考。在设计过程中，可以根据自己设计的进程与需要随时进行调整。

1. 系统用户表（Users），登录系统时使用。表结构如表9-1所示。

表9-1 系统用户表结构

| 名 | 类 型 | 允许空值 |
|---|---|---|
| UserID | Char（11） | 否 |
| UserName | Varchar（20） | 是 |
| UserKey | varchar（20） | 是 |

2. 菜谱分类表（MenuTypeTbl），为了方便点餐，将菜单信息进行分类，如热菜、凉菜等。表结构如表9-2所示。

表9-2 菜谱分类表结构

| 名 | 类 型 | 允许空值 |
|---|---|---|
| name | varchar（20） | 是 |
| id | char（11） | 否 |

3. 餐桌表（TableTbl），用来表示顾客在哪个桌位就座。表结构如表9-3所示。

表9-3　餐桌表

| 名 | 类　型 | 允许空值 |
|---|---|---|
| tableid | char（11） | 否 |
| num | char（11） | 是 |
| flag | char（11） | 是 |

4. 菜谱表（MenuTbl），菜谱表用来保存菜信息，如分类、名称和价格等。表结构如表9-4所示。

表9-4　菜谱表结构

| 名 | 类　型 | 允许空值 |
|---|---|---|
| typeID | char（11） | 是 |
| name | varchar（50） | 是 |
| pic | varchar（100） | 是 |
| price | char（11） | 是 |

5. 订单表（OderTbl），当顾客开桌点菜时生成的信息，保存到该表。表结构如表9-5所示。

表9-5　订单表结构

| 名 | 类　型 | 允许空值 |
|---|---|---|
| orderTime | varchar（30） | 是 |
| orderID | int（11） | 是 |
| userID | int（11） | 是 |
| tableID | int（11） | 是 |
| personNum | int（11） | 是 |
| isPay | int（11） | 是 |

6. 订单明细表（OrderDetailTbl），订单明细表和上面的订单表是一对多的关系，即一桌客人点了多份菜。表结构如表9-6所示。

表9-6　订单明细表结构

| 名 | 类　型 | 允许空值 |
|---|---|---|
| orderID | int（11） | 是 |
| foodID | int（11） | 是 |
| foodnum | int（11） | 是 |
| foodnote | Varchar（30） | 是 |

至此，无线点餐系统设计全部完成。

# 参 考 文 献


[1] 黄智伟，等．ARM9嵌入式系统设计基础教程［M］．北京：北京航空航天大学出版社，2008.
[2] 杜春雷．ARM体系结构与编程［M］．北京：清华大学出版社，2008.
[3] 汪成义，等．ARM嵌入式技术原理与应用［M］．北京：北京航空航天大学出版社，2011.
[4] 邱铁．ARM嵌入式系统结构与编程［M］．北京：清华大学出版社，2009.
[5] 杨铸，等．深入浅出：嵌入式底层软件开发［M］．北京：北京航空航天大学出版社，2011.
[6] 周立功．ARM嵌入式系统系列教程•ARM嵌入式系统实验教程2［M］．北京：北京航空航天大学出版社，2005.
[7] 周立功．ARM嵌入式系统软件开发实例2［M］．北京：北京航空航天大学出版社，2006.
[8] 刘洪涛．ARM嵌入式体系结构与接口技术［M］．北京：人民邮电出版社，2009.
[9] 钟睿，等．基于ARM的嵌入式Windows CE系统高级开发技术［M］．北京：清华大学出版社，2011.
[10] 长高科技．Windows CE 输出入装置与驱动程式设计-（S3C6410 WinCE 6.0 开发应用实务）［M］．台北：长高科技图书，2010.
[11] 张勇，等．Windows CE应用程序设计［M］．西安：西安电子科技大学出版社，2008.
[12] 韩少云，等．ARM嵌入式系统移植实战开发［M］．北京：北京航空航天大学出版社，2012.
[13] 徐成，等．Windows CE内核定制与驱动程序开发［M］．北京：中国电力出版社，2011.
[14] 李大为．Windows CE工程实践完全解析［M］．北京：中国电力出版社，2008.
[15] 周建设．Windows CE设备驱动及BSP开发指南［M］．北京：中国电力出版社，2009.
[16] 冯(Samuel Phung)，等．Windows CE 6.0嵌入式高级编程［M］．北京：清华大学出版社，2009.
[17] 邱小平，等．Windows CE 6开发经典［M］．北京：电子工业出版社，2009.
[18] 张欢，等．Windows CE系统开发基础与实例［M］．北京：中国电力出版社，2010.
[19] 华清远见嵌入式培训中心，等．Windows CE嵌入式开发标准教程［M］．北京：人民邮电出版社，2009.
[20] 汪兵．Windows CE嵌入式高级编程及其实例详解（用C++实现）［M］．北京：中国水利水电出版社，2008.

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "V0lORE9XUyBDReW1jOWFpeW8j+ezu+e7n+eoi+W6j+W8gOWPke+8iOWfuuS6jkFSTTEx77yJXzEzMjM2MTI0LnppcA==",
  "filename_decoded": "WINDOWS CE\u5d4c\u5165\u5f0f\u7cfb\u7edf\u7a0b\u5e8f\u5f00\u53d1\uff08\u57fa\u4e8eARM11\uff09_13236124.zip",
  "filesize": 107378801,
  "md5": "71466240c0624fe7a897504a42c8702e",
  "header_md5": "6d19d2e615a5e73995a808062a0e056a",
  "sha1": "9959df0dc459d9be0c12fdd674f30d23134b9692",
  "sha256": "0013913852343c1cf2c9bde94e00119634609eed88ecb562085e6cb42d046c5f",
  "crc32": 3128833320,
  "zip_password": "",
  "uncompressed_size": 116217955,
  "pdg_dir_name": "WINDOWS CE\u255f\u2562\u255a\u03b4\u2569\u255c\u2567\u2561\u2550\u2502\u2502\u2560\u2568\u2265\u2510\u00ac\u2556\u00f3\u00fa\u00bf\u2557\u2219\u2559\u250cARM11\u00fa\u2310_13236124",
  "pdg_main_pages_found": 472,
  "pdg_main_pages_max": 472,
  "total_pages": 484,
  "total_pixels": 2851319880,
  "pdf_generation_missing_pages": false
}